

## **Clase 15: Excepciones**

### **¿Qué es una excepción?**

Una excepción es cualquier condición de error o comportamiento inesperado que encuentra un programa en ejecución. Las excepciones pueden iniciarse debido a un error en el código propio o en el código al que se llama (por ejemplo, una biblioteca compartida), a recursos del sistema operativo no disponibles, a condiciones inesperadas que encuentra el runtime (por ejemplo, imposibilidad de comprobar el código), etc. La aplicación puede recuperarse de algunas de estas condiciones, pero no de todas. Aunque es posible recuperarse de la mayoría de las excepciones que se producen en la aplicación, no ocurre lo mismo con las excepciones de runtime.

En .NET, una excepción es un objeto que hereda de la clase `System.Exception`. Una excepción se inicia desde un área del código en la que ha producido un problema. La excepción se pasa hacia arriba en la pila hasta que la aplicación la controla o el programa finaliza.

### **Excepciones vs. métodos tradicionales de control de errores**

Tradicionalmente, el modelo de control de errores de un lenguaje se basaba en el modo exclusivo del lenguaje de detectar los errores y buscarles controladores, o bien en el mecanismo de control de errores ofrecido por el sistema operativo. La forma en que .NET implementa el control de excepciones proporciona las siguientes ventajas:

- La administración e iniciación de excepciones funciona de igual modo para los lenguajes de programación. NET.
- No requiere ninguna sintaxis de lenguaje específica para controlar las excepciones, pero permite que cada lenguaje defina su propia sintaxis.
- Las excepciones pueden iniciarse en varios procesos en incluso límites de máquina.
- Se puede agregar el código de control de excepciones a una aplicación para aumentar la confiabilidad del programa.

Las excepciones ofrecen ventajas sobre otros métodos de notificación de errores, por ejemplo, los códigos de retorno. Los errores no pasan desapercibidos porque si se inicia una excepción y no la controla, el runtime finaliza la aplicación. Los valores no válidos no continúan propagándose por el sistema como resultado de que el código no pueda encontrar un código de retorno de error.

Excepciones más comunes:

Tipo de excepción	Descripción	Ejemplo
Exception	Clase base de todas las excepciones.	Ninguno (utilice una clase derivada de esta excepción).
IndexOutOfRangeException	El tiempo de ejecución la genera solo cuando una matriz no está correctamente indexada.	La indexación de una matriz fuera de su intervalo válido: <code>arr[arr.Length+1]</code>
NullReferenceException	El tiempo de ejecución la genera solo cuando se hace referencia a un objeto null.	<code>object o = null;</code> <code>o.ToString();</code>
InvalidOperationException	Los métodos la generan si se produce un estado no válido.	Llamada a <code>Enumerator.MoveNext()</code> después de quitar un elemento de la colección subyacente.
ArgumentException	Clase base de todas las excepciones de argumento.	Ninguno (utilice una clase derivada de esta excepción).
ArgumentNullException	Los métodos que no permiten que un argumento sea null la generan.	<code>String s = null;</code> <code>"Calculate".IndexOf(s);</code>
ArgumentOutOfRangeException	Los métodos que comprueban que los argumentos se encuentran en un intervalo determinado la generan.	<code>String s = "string";</code> <code>s.Substring(s.Length+1);</code>

### ¿Qué es el manejo de excepciones (exception handling)?

El manejo de excepciones es una técnica de programación que permite al programador controlar los errores ocasionados durante la ejecución de un programa informático. Cuando ocurre cierto tipo de error, el sistema reacciona ejecutando un fragmento de código que resuelve la situación, por ejemplo retornando un mensaje de error o devolviendo un valor por defecto.

El manejo de excepciones ayuda al programador a trasladar el código para manejo de errores de la línea principal de ejecución, además se puede elegir entre manejar todas las excepciones, las de cierto tipo o de las de grupos relacionados, esto hace que la probabilidad de pasar por alto los errores se reduzca y a la vez hace los programas más robustos. Pero es importante utilizar un lenguaje de programación que soporte este manejo, de lo contrario el procesamiento de errores no estará incluido y hará el programa más vulnerable. Este manejo está diseñado para procesar errores que ocurren cuando se ejecuta una instrucción, algunos ejemplos son: desbordamiento aritmético, división entre cero, parámetros inválidos de método y asignación fallida en la memoria. Sin embargo, no está diseñado para procesar problemas con eventos independientes al programa como son pulsar una tecla o clic al mouse.

Las características de control de excepciones del lenguaje C# le ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. El control de excepciones usa las palabras clave `try`, `catch` y `finally` para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante. Las excepciones las puede generar Common Language Runtime (CLR), .NET, bibliotecas de terceros o el código de aplicación. Las excepciones se crean mediante el uso de la palabra clave `throw`.

En muchos casos, una excepción la puede no producir un método al que el código ha llamado directamente, sino otro método más bajo en la pila de llamadas. Cuando se genera una excepción, CLR desenreda la pila, busca un método con un bloque `catch` para el tipo de excepción específico y ejecuta el

primer bloque catch que encuentra. Si no encuentra ningún bloque catch adecuado en cualquier parte de la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

En este ejemplo, un método prueba a hacer la división entre cero y detecta el error. Sin el control de excepciones, este programa finalizaría con un error `DivideByZeroException` no controlada.

```
C#

public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

### ¿Qué son las excepciones en tiempo de ejecución y qué tienen en común? (técnicamente)

Son errores que se presentan cuando la aplicación se está ejecutando. Su origen puede ser diverso, se pueden producir por un uso incorrecto del programa por parte del usuario (si ingresa una cadena cuando se espera un número), o se pueden presentar debido a errores de programación (acceder a localidades no reservadas o hacer divisiones entre cero), o debido a algún recurso externo al programa (al acceder a un archivo o al conectarse a algún servidor o cuando se acaba el espacio en la pila (stack) de la memoria).

### ¿Qué es el Stack Trace o pila de llamadas? ¿En qué orden se lee?

Una **pila de llamadas** es un mecanismo para que un intérprete (como el intérprete de JavaScript en un navegador web) realice un seguimiento de en qué lugar se llama a múltiples funciones, qué función se está ejecutando actualmente y qué funciones son llamadas desde esa función, etc.

- Cuando un script llama a una función, el intérprete la añade a la pila de llamadas y luego empieza a ejecutar la función.
- Cualquier función o funciones que sean llamadas por esa función son añadidas arriba de la pila de llamadas y serán ejecutadas cuando su llamada sea alcanzada.
- Cuando la función actual termina, el intérprete la elimina de la pila y reanuda la ejecución donde se quedó.
- Si la pila necesita más espacio del que se le asignó, se producirá un error de "desbordamiento de pila".

## ¿Cuál es la función de cada bloque dentro de una estructura try-catch?

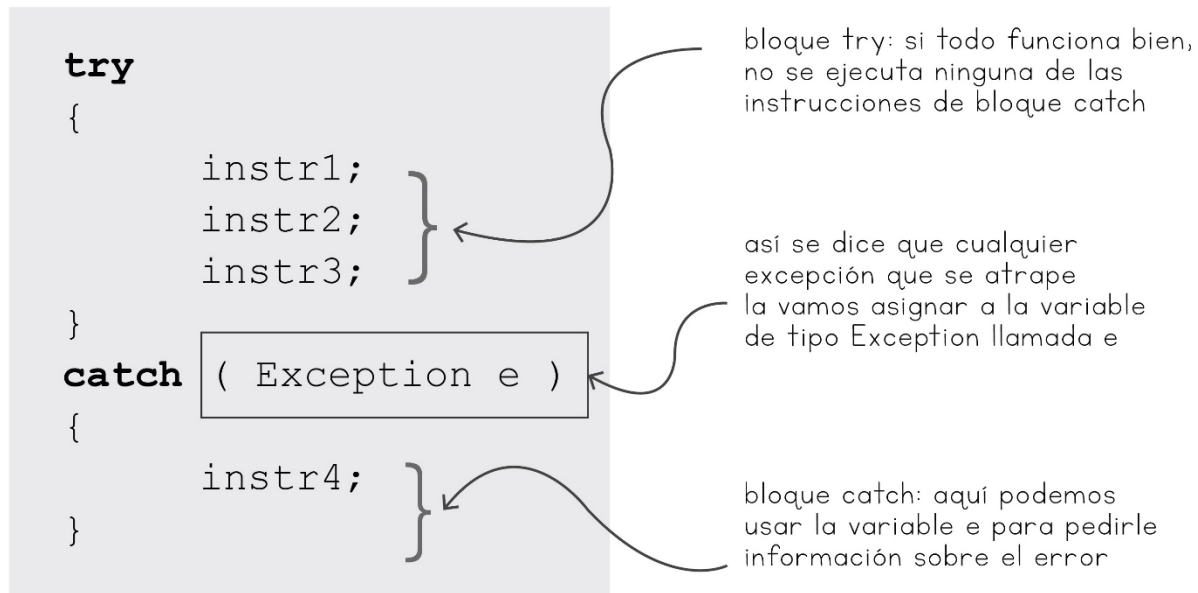
Coloque las instrucciones de código que podrían elevar o producir una excepción en un bloque try, y las que se usan para controlar la excepción o excepciones en uno o varios bloques catch debajo del bloque try. Cada bloque catch incluye el tipo de excepción y puede contener instrucciones adicionales necesarias para controlar ese tipo de excepción.

Common Language Runtime (CLR) detecta las excepciones no controladas por los bloques catch. Si CLR detecta una excepción, puede producirse uno de los resultados siguientes, en función de la configuración de CLR:

- Aparece un cuadro de diálogo Depurar.
- El programa detiene la ejecución y aparece un cuadro de diálogo con información de la excepción.
- Se imprime un error en el flujo de salida de error estándar.

El bloque try contiene una expresión que puede generar la excepción. En caso de producirse la excepción, el runtime detiene la ejecución normal y empieza a buscar un bloque catch que pueda capturar la excepción pendiente (basándose en su tipo). Si en la función inmediata no se encuentra un bloque catch adecuado, el runtime desenreda la pila de llamadas en busca de la función de llamada. Si tampoco ahí encuentra un bloque catch apropiado, busca la función que llamó a la función de llamada y así sucesivamente hasta encontrar un bloque catch (o hasta llegar al final, en cuyo caso se cerrará el programa).

Si encuentra un bloque catch, se considera que la excepción ha sido capturada y se reanuda la ejecución normal desde el cuerpo del bloque catch (que, en el caso de la diapositiva, escribe el mensaje contenido en el objeto excepción OverflowException). Por lo tanto, el uso de bloques try-catch hace que las instrucciones para tratamiento de errores no se mezclen con las instrucciones lógicas básicas, por lo que el programa es más fácil de interpretar.



**En el caso de una misma estructura try-catch que tiene más de un tipo de bloque catch, ¿Se podría ejecutar más de un bloque catch? ¿Por qué?**

Un bloque de código en una instancia try puede contener muchas instrucciones, cada una de las cuales puede producir una o más clases diferentes de excepción. Al haber muchas clases de excepciones distintas, es posible que haya muchos bloques catch y que cada uno de ellos capture un tipo específico

de excepción. La captura de una excepción se basa únicamente en su tipo. El runtime captura automáticamente objetos excepción de un tipo concreto en un bloque catch para ese tipo.

### **¿En qué parte del código continúa la ejecución del programa una vez manejada una excepción?**

Cuando se produce una excepción, se detiene la ejecución y se proporciona el control al controlador de excepciones adecuado. A menudo, esto significa que se omiten líneas de código que esperaba que se ejecuten. Se debe realizar alguna limpieza de recursos, como cerrar un archivo, aunque se inicie una excepción. Para ello, puede usar un bloque finally. Un bloque finally siempre se ejecuta, independientemente de si se inicia una excepción.

El programa continua en la línea siguiente al bloque try-catch del código.

### **¿Existe una forma de capturar cualquier excepción sin importar su tipo? ¿Qué habría que considerar en ese caso si se tiene más de un tipo de bloque catch en la misma estructura?**

Un bloque catch general (Exception), puede capturar cualquier excepción independientemente de su clase y se utiliza con frecuencia para capturar cualquier posible excepción que se pudiera producir por la falta de un controlador adecuado. Un bloque try no puede tener más que un bloque catch general. En caso de existir, un bloque catch general debe ser el último bloque catch en el programa.

En general, en programación es recomendable detectar un tipo de excepción específico en lugar de usar una instrucción catch básica.

Cuando se produce una excepción, se pasa a la pila y cada bloque Catch tiene la oportunidad de controlarla. El orden de las instrucciones Catch es importante. Ponga los bloques Catch dirigidos a excepciones específicas antes de un bloque Catch de excepción general o el compilador podría emitir un error. El bloque Catch adecuado se determina haciendo coincidir el tipo de la excepción con el nombre de la excepción especificada en el bloque Catch. Si no hay ningún bloque Catch específico, la excepción se detecta mediante un bloque Catch general, si existe alguno.

### **¿Qué sucede cuando se lanza una excepción? ¿Qué sucede si no la manejo / controlo?**

Puede iniciar explícitamente una excepción mediante la instrucción throw. También se puede iniciar una excepción detectada usando de nuevo la instrucción throw. En diseño de código, es recomendable agregar información a una excepción que se vuelve a iniciar para proporcionar más información durante la depuración.

Cuando necesita lanzar una excepción, el runtime ejecuta una instrucción throw y lanza una excepción definida por el sistema. Esto interrumpe inmediatamente la secuencia de ejecución normal del programa y transfiere el control al primer bloque catch que pueda hacerse cargo de la excepción en función de su clase. Es posible utilizar la instrucción throw para lanzar excepciones propias. Pueden generar excepciones Common Language Runtime (CLR), .NET Framework, las bibliotecas de otros fabricantes o el código de aplicación.

### **¿Cómo lanzo una excepción?**

Para lanzar una excepción debemos utilizar la palabra clave 'throw' y seguidamente la referencia de un objeto de la clase 'Exception' o de una subclase de 'Exception':

**Dentro de un bloque catch, ¿cuál es la diferencia entre "throw;" y "throw ex;"? (ex es un identificador para la instancia de una excepción capturada)**

throw: Si usamos la instrucción "throw", conserva la información original de la pila de errores. En el manejo de excepciones, " throw " con un parámetro vacío también se llama volver a lanzar la última excepción.

throw ex: Si usamos la instrucción "throw ex", el seguimiento de la pila de la excepción será reemplazado por un seguimiento de la pila comenzando en el punto de relanzamiento. Se utiliza para ocultar intencionalmente la información de seguimiento de la pila.

**¿Las excepciones se lanzan en tiempo de compilación o de ejecución? ¿Por qué?**

Se lanzan en tiempo de ejecución ya que el tipo de errores que pueden capturar solo pueden ocurrir en tiempo de ejecución.

**¿Cómo creo una excepción propia?**

.NET contiene muchas excepciones distintas que puede usar. Sin embargo, en algunos casos, cuando ninguna de ellas satisface sus necesidades, puede crear sus propias excepciones personalizadas.

Supongamos que desea crear un StudentNotFoundException que contiene una propiedad StudentName. Para crear una excepción personalizada, siga estos pasos:

1- Cree una clase serializable que herede de Exception. El nombre de clase debe terminar en "Exception":

```
[Serializable]
public class StudentNotFoundException : Exception { }
```

2- Agregue los constructores predeterminados:

```
C#

[Serializable]
public class StudentNotFoundException : Exception
{
    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

3- Defina cualquier propiedad y constructores adicionales:

```
C#  
  
[Serializable]  
public class StudentNotFoundException : Exception  
{  
    public string StudentName { get; }  
  
    public StudentNotFoundException() { }  
  
    public StudentNotFoundException(string message)  
        : base(message) { }  
  
    public StudentNotFoundException(string message, Exception inner)  
        : base(message, inner) { }  
  
    public StudentNotFoundException(string message, string studentName)  
        : this(message)  
    {  
        StudentName = studentName;  
    }  
}
```

4- La excepción se puede iniciar en cualquier parte con código como el siguiente:

```
throw new StudentNotFoundException("The student cannot be found.", "John");
```

**¿Qué es la propiedad InnerException? Describa a qué clase pertenece, su contenido y qué tipo de dato almacena.**

Obtiene la instancia Exception que produjo la excepción actual.

Describe el error que causó la excepción actual. La propiedad InnerException devuelve el mismo valor que se pasó al constructor Exception(String, Exception) o null si no se suministró el valor de la excepción interna al constructor. Esta propiedad es de sólo lectura. Utilice la propiedad InnerException para obtener el conjunto de excepciones que dieron lugar a la excepción actual.

**De existir un bloque finally. ¿Bajo qué condiciones se ejecutará el código que contiene? ¿Dónde se ubica el bloque finally dentro de una estructura de manejo de excepciones?**

Cuando se produce una excepción, se detiene la ejecución y se proporciona el control al controlador de excepciones adecuado. A menudo, esto significa que se omiten líneas de código que esperaba que se ejecuten. Se debe realizar alguna limpieza de recursos, como cerrar un archivo, aunque se inicie una excepción. Para ello, puede usar un bloque finally. Un bloque finally siempre se ejecuta, independientemente de si se inicia una excepción.

La cláusula finally de C# contiene un conjunto de instrucciones que es necesario ejecutar sea cual sea el flujo de control. Las instrucciones del bloque finally se ejecutarán aunque el control abandone un bucle try como resultado de la ejecución normal porque el flujo de control llega al final del bloque try. Del mismo modo, también se ejecutarán las instrucciones del bloque finally si el control abandona un bucle try como resultado de una instrucción throw o una instrucción de salto como break o continue. El bloque finally es útil en dos casos: para evitar la repetición de instrucciones y para liberar recursos tras el lanzamiento de una excepción.

Ejemplo:

```
try
{
    // Código
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}
catch (DivideByZeroException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("Pulse una tecla para continuar...");
}
```

## **Clase 16: Test Unitarios:**

### **¿Qué es una prueba unitaria?**

- Comprueban la funcionalidad específica de una pequeña parte de una aplicación, como clases, métodos
- Cada prueba es independiente del resto
- Tendremos tantas como sea necesario para probar el 100% (o lo máximo posible) de los posibles casos que estén contemplados en el código.
- Mejoran la lectura del código al tener ejemplos de uso

### **¿Qué es una prueba integral?**

- Verifican que las relaciones existentes entre las diferentes clases y servicios funcionan correctamente.
- Se busca encontrar todos esos problemas que surgen al juntar componentes desarrollados de forma independiente.
- por ejemplo, pueden probar la integración de la lógica de negocio que hizo un desarrollador con las conexiones a la base de datos que hizo otro.

### **¿Qué es una prueba funcional?**

- Las pruebas funcionales verifican el comportamiento del sistema para afirmar que se cumple con la funcionalidad completa. Se les suele denominar también pruebas End-To-End o E2E.
- Se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.
- Verifican el flujo completo de la aplicación.
- Según su ejecución pueden ser Manuales o Automatizadas.

### **¿Qué es el patrón AAA? Describa cada una de sus etapas.**

El patrón sugiere dividir una prueba unitaria (un método de pruebas) en tres secciones:

- Arrange: Inicializa los objetos y establece los valores de los datos que vamos a utilizar en el test.
- Act: Realiza la llamada al método a probar con los parámetros preparados para tal fin.
- Assert: Comprueba que el método ejecutado se comporta tal y como teníamos previsto que lo hiciera.



Ejemplo:

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);
    // act
    account.Withdraw(withdrawal);
    double actual = account.Balance;
    // assert
    Assert.AreEqual(expected, actual);
}
```

### ¿Qué es la clase Assert? ¿Para qué sirve?

Clase Assert:

- Define una serie de métodos estáticos que analizan una condición True - False
- Determina si la prueba se supera o no:
  - o AreEqual()
  - o AreNotEqual
  - o IsFalse()
  - o IsTrue()
  - o IsNull()
  - o IsNotNull()
- El Assert también puede ser manejado desde los atributos o etiquetas del método, por ejemplo para comprobar si se produjo una excepción.

### Clase 17: Tipos Genéricos

#### ¿Qué es una clase genérica? ¿Qué permite?

Generics introduce el concepto de parámetros de tipo en .NET, lo que hace posible diseñar clases y métodos que difieran la especificación de uno o más tipos hasta que la clase o método sea declarado y instanciado por el código del cliente. Por ejemplo, al usar un parámetro de tipo genérico T, puede escribir una sola clase que otro código de cliente puede usar sin incurrir en el costo o el riesgo de las operaciones de conversión en tiempo de ejecución u operaciones de encapsulamiento.

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos determinado. El uso más común de las clases genéricas es con colecciones como listas vinculadas, tablas hash, pilas, colas y árboles, entre otros. Las operaciones como la adición y eliminación de elementos de la colección se realizan básicamente de la misma manera independientemente del tipo de datos que se almacenan.

Normalmente, crea clases genéricas empezando con una clase concreta existente, y cambiando tipos en parámetros de tipo de uno en uno hasta que alcanza el equilibrio óptimo de generalización y facilidad de uso. Al crear sus propias clases genéricas, entre las consideraciones importantes se incluyen las siguientes:

- Los tipos que se van a generalizar en parámetros de tipo.

Como norma, cuantos más tipos pueda parametrizar, más flexible y reutilizable será su código. En cambio, demasiada generalización puede crear código que sea difícil de leer o entender para otros desarrolladores.

- Las restricciones, si existen, que se van a aplicar a los parámetros de tipo.

Una buena norma es aplicar el máximo número de restricciones posible que todavía le permitan tratar los tipos que debe controlar. Por ejemplo, si sabe que su clase genérica está diseñada para usarse solo con tipos de referencia, aplique la restricción de clase. Esto evitará el uso no previsto de su clase con tipos de valor, y le permitirá usar el operador `as` en `T`, y comprobar si hay valores `NULL`.

- Si separar el comportamiento genérico en clases base y subclasses.

Como las clases genéricas pueden servir como clases base, las mismas consideraciones de diseño se aplican aquí con clases no genéricas. Vea las reglas sobre cómo heredar de clases base genéricas posteriormente en este tema.

- Si implementar una o más interfaces genéricas.

Por ejemplo, si está diseñando una clase que se usará para crear elementos en una colección basada en genéricos, puede que tenga que implementar una interfaz como `IComparable<T>` donde `T` es el tipo de su clase.

### ¿Qué es una restricción o constraint de un tipo genérico? ¿Qué sucede si no hay restricciones?

La cláusula `where` en una definición genérica especifica restricciones en los tipos que se usan como argumentos para los parámetros de tipo en un tipo genérico, método, delegado o función local. Las restricciones pueden especificar interfaces o clases base, o bien requerir que un tipo genérico sea una referencia, un valor o un tipo no administrado. Declaran las funcionalidades que debe tener el argumento de tipo.

a cláusula `where` también puede incluir una restricción de clase base. La restricción de clase base indica que un tipo que se va a usar como argumento de tipo para ese tipo genérico tiene la clase especificada como clase base, o bien es la clase base. Si se usa la restricción de clase base, debe aparecer antes que cualquier otra restricción de ese parámetro de tipo.

**Complete la tabla:**

<code>where T struct</code>	:	El argumento de tipo debe ser un tipo de valor que no acepta valores <code>NULL</code> . Para más información sobre los tipos de valor que admiten un valor <code>NULL</code> , consulte Tipos de valor que admiten un valor <code>NULL</code> . Todos los tipos de valor tienen un constructor sin parámetros accesible, por lo que la restricción <code>struct</code> implica la restricción <code>new()</code> y no se puede combinar con la restricción <code>new()</code> . No puede combinar la restricción <code>struct</code> con la restricción <code>unmanaged</code> .
<code>where T : class</code>	:	El argumento de tipo debe ser un tipo de referencia. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz. En un contexto que admite un valor <code>NULL</code> en C# 8.0 o versiones posteriores, <code>T</code> debe ser un tipo de referencia que no acepte valores <code>NULL</code> .
<code>where T class?</code>	:	El argumento de tipo debe ser un tipo de referencia, que acepte o no valores <code>NULL</code> . Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz.
<code>where T notnull</code>	:	El argumento de tipo debe ser un tipo que no acepta valores <code>NULL</code> . El argumento puede ser un tipo de referencia que no acepta valores <code>NULL</code> en C# 8.0 o posterior, o bien un tipo de valor que no acepta valores <code>NULL</code> .

where T default	:	Esta restricción resuelve la ambigüedad cuando es necesario especificar un parámetro de tipo sin restricciones al invalidar un método o proporcionar una implementación de interfaz explícita. La restricción default implica el método base sin la restricción class o struct. Para obtener más información, vea la propuesta de especificación de la restricción default.
where T unmanaged	:	El argumento de tipo debe ser un tipo no administrado que no acepta valores NULL. La restricción unmanaged implica la restricción struct y no se puede combinar con las restricciones struct ni new().
where T new()	:	El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción new() debe especificarse en último lugar. La restricción new() no se puede combinar con las restricciones struct ni unmanaged.
where : <base class name>	T	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, T debe ser un tipo de referencia que no acepta valores NULL derivado de la clase base especificada.
where : <base class name>?	T	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, T puede ser un tipo que acepta o no acepta valores NULL derivado de la clase base especificada.
where : <interface name>	T	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, T debe ser un tipo que no acepta valores NULL que implementa la interfaz especificada.
where : <interface name>?	T	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica. En un contexto que admite un valor NULL en C# 8.0, T puede ser un tipo de referencia que admite un valor NULL, un tipo de referencia que no acepta valores NULL o un tipo de valor. T no puede ser un tipo de valor que admite un valor NULL.
where T : U		El argumento de tipo proporcionado por T debe ser o se debe derivar del argumento proporcionado para U. En un contexto que admite un valor NULL, si U puede ser un tipo de referencia que no acepta valores NULL, T debe ser un tipo de referencia que no acepta valores NULL. Si U es un tipo de referencia que admite un valor NULL, T puede aceptar valores NULL o no.

## **Clase 18 – Interfaces:**

### **¿Qué es una interfaz? ¿Qué implica implementarla en una clase?**

Una interfaz define un contrato. Cualquier class o struct que implemente ese contrato debe proporcionar una implementación de los miembros definidos en la interfaz.

- Son una manera de describir qué debería hacer una clase sin especificar el cómo.
- Es la descripción de uno o más métodos que posteriormente alguna clase puede implementar.
- C# no permite especificar atributos en las interfaces.
- Todos los métodos son públicos (no se permite especificarlo).
- Todos los métodos son como “abstractos” ya que no cuentan con implementación (no se permite especificarlo).
- Se pueden especificar propiedades (sin implementación).
- Las clases pueden implementar varias interfaces.
- Las interfaces pueden “simular” algo parecido a la herencia múltiple.

## ¿Qué miembros puede especificar una interfaz? (atributos, métodos, propiedades, etc)

Una interfaz puede incluir:

- Constantes
- Operadores
- Constructor estático.
- Tipos anidados
- Campos, métodos, propiedades, indizadores y eventos estáticos.
- Declaraciones de miembros con la sintaxis de implementación de interfaz explícita.
- Modificadores de acceso explícitos (el acceso predeterminado es public).

## ¿Una interfaz puede implementar otra interfaz?

Una interfaz puede heredar de una o varias interfaces base.

## ¿Qué significa implementar una interfaz de forma explícita? ¿Qué utilidad tiene? ¿Qué consecuencias o efectos tiene?

- Los miembros implementados explícitamente sirven para ocultar la implementación de miembros de interfaces a las clases que lo implementan.
- También sirve para evitar la ambigüedad cuando, por ejemplo, una clase implementa dos interfaces las cuales poseen un miembro con la misma firma.
- Las clases derivadas de clases que implementan interfaces de manera explícita no pueden sobrescribir los métodos definidos explícitamente.
- Sintácticamente la implementación de una interfaz de manera explícita e implícita es igual, lo único que cambia es la firma del miembro en la clase que implementa la interfaz.



```
public class Cuervo :IMensaje, IEncriptado
{
    public string EnviarMensaje()
    {
        return "Llego el invierno;";
    }

    string IEncriptado.EnviarMensaje()
    {
        return "Jon Snow es el verdadero rey;";
    }
}
```

Se coloca el nombre de la interfaz adelante del nombre del método que se está implementando explícitamente y además no se le indica la visibilidad.

## ¿Se pueden tener interfaces genéricas? ¿Se puede restringir sus parámetros de tipo? ¿Puede implementarlas cualquier clase o sólo clases que también sean genéricas?

Las interfaces genéricas proporcionan homólogas con seguridad de tipos a las interfaces no genéricas para realizar comparaciones de ordenación y de igualdad, y para obtener funcionalidad compartida por los tipos de colección genéricos.

## ¿En qué se diferencian una interfaz y una clase abstracta?

Existen varias diferencias entre una clase abstracta y una interfaz:

- 1- Una clase abstracta puede heredar o extender cualquier clase (independientemente de que esta sea abstracta o no), mientras que una interfaz solamente puede extender o implementar otras interfaces.
- 2- Una clase abstracta puede heredar de una sola clase (abstracta o no) mientras que una interfaz puede extender varias interfaces de una misma vez.

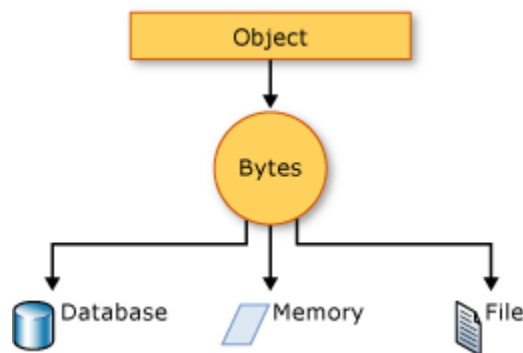
- 3- Una clase abstracta puede tener métodos que sean abstractos o que no lo sean, mientras que las interfaces sólo y exclusivamente pueden definir métodos abstractos.
- 4- En java concretamente (ya que has puesto la etiqueta Java), en las clases abstractas la palabra abstract es obligatoria para definir un método abstracto (así como la clase). Cuando defines una interfaz, esta palabra es opcional ya que se infiere en el concepto de interfaz.
- 5- En una clase abstracta, los métodos abstractos pueden ser public o protected. En una interfaz solamente puede haber métodos públicos.

### **Clase 19 – Archivos y Serialización:**

**¿Qué es serializar? ¿Para qué sirve serializar? ¿En qué formatos se puede serializar (vistos en clase)? Indique qué miembros de la clase se incluyen en cada formato. Indique qué características debe tener la clase para ser serializable en cada formato.**

La serialización es el proceso de convertir un objeto en una secuencia de bytes para almacenarlo o transmitirlo a la memoria, a una base de datos o a un archivo. Su propósito principal es guardar el estado de un objeto para poder volver a crearlo cuando sea necesario. El proceso inverso se denomina deserialización.

Funcionamiento de la serialización:



El objeto se serializa en una secuencia que incluye los datos. La secuencia también puede tener información sobre el tipo del objeto, como la versión, la referencia cultural y el nombre del ensamblado. A partir de esa secuencia, el objeto se puede almacenar en una base de datos, en un archivo o en memoria.

Usos de la serialización:

La serialización permite al desarrollador guardar el estado de un objeto y volver a crearlo según sea necesario, ya que proporciona almacenamiento de los objetos e intercambio de datos. A través de la serialización, un desarrollador puede realizar acciones como las siguientes:

- Enviar el objeto a una aplicación remota mediante un servicio web
- Pasar un objeto de un dominio a otro
- Pasar un objeto a través de un firewall como una cadena JSON o XML
- Mantener la seguridad o información específica del usuario entre aplicaciones

Serialización XML y binaria:

La serialización binaria utiliza la codificación binaria para generar una serialización compacta para usos como almacenamiento o secuencias de red basadas en socket. En la serialización binaria se serializan todos los miembros, incluso aquellos que son de solo lectura, y mejora el rendimiento.

La serialización XML serializa las propiedades y los campos públicos de un objeto o los parámetros y valores devueltos de los métodos en una secuencia XML que se ajusta a un documento específico del lenguaje de definición de esquema XML (XSD). La serialización XML produce clases fuertemente tipadas cuyas propiedades y campos públicos se convierten a XML. System.Xml.Serialization contiene clases para serializar y deserializar XML. Se aplican atributos a clases y a miembros de clase para controlar la forma en que XmlSerializer serializa o deserializa una instancia de la clase.

Conversión de un objeto en serializable:

Para la serialización binaria o XML, necesita lo siguiente:

- Objeto que se va a serializar
- Secuencia para incluir el objeto serializado
- Instancia de System.Runtime.Serialization.Formatter

Serialización XML:

- La serialización XML sólo serializa los atributos públicos y los valores de propiedad de un objeto en una secuencia XML.
- La serialización XML no convierte los métodos, indexadores, atributos privados ni propiedades de sólo lectura (salvo colecciones de sólo lectura).
- La clase central de la serialización XML es XmlSerializer y sus métodos más importantes son Serialize y Deserialize.
- La secuencia XML que genera XmlSerializer cumple con la recomendación 1.0 del W3C ([www.w3.org](http://www.w3.org)) acerca del lenguaje de definición de esquemas XML (XSD).
- Además, los tipos de datos generados cumplen las especificaciones enumeradas en el documento titulado "XML Schema Part 2: Datatypes".

Al crear una aplicación que utiliza la clase XmlSerializer, debe tener en cuenta los siguientes elementos y sus implicaciones:

- La clase XmlSerializer crea archivos C# (.cs) y los compila en archivos .dll en el directorio especificado por la variable de entorno TEMP; la serialización se produce con esos archivos DLL.
- Una clase debe tener un constructor por defecto para que XmlSerializer pueda serializarla.
- Sólo se pueden serializar los atributos y propiedades públicas.
- Los métodos no se pueden serializar.

### **¿Qué es una tabla? Describa su composición.**

Las tablas son objetos de base de datos que contienen todos sus datos. En las tablas, los datos se organizan con arreglo a un formato de filas y columnas, similar al de una hoja de cálculo. Cada fila representa un registro único y cada columna un campo dentro del registro. Por ejemplo, en una tabla que contiene los datos de los empleados de una compañía puede haber una fila para cada empleado y distintas columnas en las que figuren detalles de los mismos, como el número de empleado, el nombre, la dirección, el puesto que ocupa y su número de teléfono particular.

El número de tablas de una base de datos se limita solo por el número de objetos admitidos en una base (2.147.483.647). Una tabla definida por el usuario estándar puede tener hasta 1.024 columnas. El número de filas de la tabla solo está limitado por la capacidad de almacenamiento del servidor.

Puede asignar propiedades a la tabla y a cada columna de la tabla para controlar los datos admitidos y otras propiedades. Por ejemplo, puede crear restricciones en una columna para no permitir valores nulos o para proporcionar un valor predeterminado si no se especifica un valor, o puede asignar una restricción de clave en la tabla que exige la unicidad o definir una relación entre las tablas.

**¿Qué es una primary key o clave primaria? Describa sus características.**

**¿Qué es una foreign key o clave foránea?**

En el diseño de bases de datos relacionales, se llama clave primaria o clave principal a un campo o a una combinación de campos que identifica de forma única a cada fila de una tabla. Una clave primaria comprende de esta manera una columna o conjunto de columnas. No puede haber dos filas en una tabla que tengan la misma clave primaria.

Ejemplos de claves primarias son DNI (asociado a una persona) o ISBN (asociado a un libro). Las guías telefónicas y diccionarios no pueden usar nombres o palabras o números del sistema decimal de Dewey como claves candidatas, porque no identifican unívocamente números de teléfono o palabras.

El modelo relacional, según se lo expresa mediante cálculo relacional y álgebra relacional, no distingue entre clave primaria y otros tipos de claves. Las claves primarias fueron agregadas al estándar SQL principalmente para conveniencia del programador. En un modelo entidad-relación, la clave primaria permite las relaciones de la tabla que tiene la clave primaria con otras tablas que van a utilizar la información de esta tabla.

Tanto claves únicas como claves primarias pueden referenciarse con claves foráneas.

Limitaciones y restricciones:

- Una tabla solo puede incluir una restricción PRIMARY KEY.
- Todas las columnas definidas en una restricción PRIMARY KEY se deben definir como NOT NULL. Si no se especifica nulabilidad, la nulabilidad de todas las columnas que participan en una restricción PRIMARY KEY se establece en NOT NULL.

En el contexto de bases de datos relacionales, una clave foránea o clave ajena (o Foreign Key FK) es una limitación referencial entre dos tablas. La clave foránea identifica una columna o grupo de columnas en una tabla (tabla hija o referendo) que se refiere a una columna o grupo de columnas en otra tabla (tabla maestra o referenciada). Las columnas en la tabla referendo deben ser la clave primaria u otra clave candidata en la tabla referenciada.

Los valores en una fila de las columnas referendo deben existir solo en una fila en la tabla referenciada. Así, una fila en la tabla referendo no puede contener valores que no existen en la tabla referenciada. De esta forma, las referencias pueden ser creadas para vincular o relacionar información. Esto es una parte esencial de la normalización de base de datos. Múltiples filas en la tabla referendo pueden hacer referencia, vincularse o relacionarse a la misma fila en la tabla referenciada. Mayormente esto se ve reflejado en una relación uno (tabla maestra o referenciada) a muchos (tabla hija o referendo).

La tabla referendo y la tabla referenciada pueden ser la misma, esto es, la clave foránea remite o hace referencia a la misma tabla. Esta clave externa es conocida en SQL:2003 como auto-referencia o clave foránea recursiva. Una tabla puede tener múltiples claves foráneas y cada una puede tener diferentes tablas referenciadas. Cada clave foránea es forzada independientemente por el sistema de base de datos. Por tanto, las relaciones en cascada entre tablas pueden realizarse usando claves foráneas. Configuraciones impropias de las claves foráneas o primarias o no forzar esas relaciones son frecuentemente la fuente de muchos problemas para la base de datos o para el modelamiento de los mismos.

Por ejemplo, digamos que hay dos tablas, una tabla CONSUMIDOR que incluye todos los datos de los consumidores, y otra que es la tabla de ÓRDENES. La intención es que todas las órdenes estén asociadas a la información del consumidor y que viven en su propia tabla. Para lograr esto debemos colocar una clave foránea en la tabla ÓRDENES con relación a la llave primaria de la tabla CONSUMIDOR.

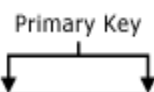
La clave foránea identifica una columna(s) en una TABLA REFERENCIANTE a una columna(s) en la TABLA REFERENCIADA.

Restricciones de clave principal:

Una tabla suele tener una columna o una combinación de columnas cuyos valores identifican de forma única cada fila de la tabla. Estas columnas se denominan claves principales de la tabla y exigen la integridad de entidad de la tabla. Debido a que las restricciones de clave principal garantizan datos únicos, con frecuencia se definen en una columna de identidad.

Cuando especifica una restricción de clave principal en una tabla, Motor de base de datos exige la unicidad de los datos mediante la creación automática de un índice único para las columnas de clave principal. Este índice también permite un acceso rápido a los datos cuando se usa la clave principal en las consultas. Si se define una restricción de clave principal para más de una columna, puede haber valores duplicados dentro de la misma columna, pero cada combinación de valores de todas las columnas de la definición de la restricción de clave principal debe ser única.

Como se muestra en la siguiente ilustración, las columnas ProductID y VendorID de la tabla Purchasing.ProductVendor forman una restricción de clave principal compuesta para esta tabla. De este modo, se garantiza que todas las filas de la tabla ProductVendor tengan una combinación de ProductID y VendorID. Esto impide la inserción de filas duplicadas.



ProductID	VendorID	AverageLeadTime	StandardPrice	LastReceiptCost
1	1	17	47.8700	50.2635
2	104	19	39.9200	41.9160
7	4	17	54.3100	57.0255
609	7	17	25.7700	27.0585
609	100	19	28.1700	29.5785

ProductVendor table

- Una tabla solo puede incluir una restricción de clave principal.
- Una clave principal no puede superar las 16 columnas y una longitud de clave total de 900 bytes.
- El índice generado por una restricción de clave principal no puede hacer que el número de índices de la tabla supere 999 índices no clúster y 1 índice clúster.
- Si no se especifica si es en clúster o no en clúster para una restricción de clave principal, se usa la disposición en clúster si no hay índices clúster en la tabla.
- Todas las columnas definidas en una restricción de clave principal se deben definir como no NULL. Si no se especifica nulabilidad, la nulabilidad de todas las columnas que participan en una restricción de clave principal se establece en no NULL.
- Si la clave principal se define en una columna de tipo definido por el usuario CLR, la implementación del tipo debe admitir el orden binario.



**¿Qué es un campo de tipo identidad en SQL Server? ¿Todas las primary keys son de tipo identidad?  
¿Puedo tener campos que no sean primary key y sean identidad?**

Una identidad o identity en SQL Server es una columna que se asigna al crear o alterar (alter) una tabla desde el diseñador o por T-SQL. Una columna como identidad es auto incrementable, especificando el incremento para cada nuevo registro.

Las columnas de identidad pueden usarse para generar valores de clave. La propiedad de identidad de una columna garantiza lo siguiente:

Cada nuevo valor se genera basándose en el valor actual de inicialización e incremento.

Cada nuevo valor de una transacción determinada es diferente de otras transacciones simultáneas de la tabla.

La propiedad de identidad de una columna no garantiza lo siguiente:

- Uniqueness of the value (Unicidad del valor): La unicidad debe aplicarse mediante una restricción PRIMARY KEY o UNIQUE, o mediante un índice UNIQUE. –
- Consecutive values within a transaction (Valores consecutivos en una transacción): No se garantiza que una transacción que inserta varias filas obtenga valores consecutivos para las filas porque podrían producirse otras inserciones simultáneas en la tabla. Si los valores deben ser consecutivos, la transacción debe usar un bloqueo exclusivo en la tabla o usar el nivel de aislamiento SERIALIZABLE.
- Consecutive values after server restart or other failures (Valores consecutivos después de un reinicio del servidor u otros errores) -SQL Server podría almacenar en memoria caché los valores de identidad por motivos de rendimiento y algunos de los valores asignados podrían perderse durante un error de la base de datos o un reinicio del servidor. Esto puede tener como resultado espacios en el valor de identidad al insertarlo. Si no es aceptable que haya espacios, la aplicación debe usar mecanismos propios para generar valores de clave. El uso de un generador de secuencias con la opción NOCACHE puede limitar los espacios a transacciones que nunca se llevan a cabo.
- Reuse of values (Reutilización de valores): Para una propiedad de identidad determinada, con un valor de inicialización e incremento específico, el motor no reutiliza los valores de identidad. Si una instrucción de inserción concreta produce un error o si la instrucción de inserción se revierte, los valores de identidad utilizados se pierden y no volverán a generarse. Esto puede tener como resultado espacios cuando se generan los valores de identidad siguientes.

**Clase 22 - Bases de datos:**

**¿Qué es ADO.NET?**

ADO.NET es un conjunto de componentes del software que pueden ser usados por los programadores para acceder a datos y a servicios de datos. Es parte de la biblioteca de clases base que están incluidas en el Microsoft .NET Framework. Es comúnmente usado por los programadores para acceder y para modificar los datos almacenados en un sistema gestor de bases de datos relacionales, aunque también puede ser usado para acceder a datos en fuentes no relacionales.

ADO.NET consiste en dos partes primarias:

Data provider:

Estas clases proporcionan el acceso a una fuente de datos, como Microsoft SQL Server y Oracle. Cada fuente de datos tiene su propio conjunto de objetos del proveedor, pero cada uno tienen un conjunto común de clases de utilidad:

- **Connection:** Proporciona una conexión usada para comunicarse con la fuente de datos. También actúa como Abstract Factory para los objetos command.
- **Command:** Usado para realizar alguna acción en la fuente de datos, como lectura, actualización, o borrado de datos relacionales.
- **Parameter:** Describe un simple parámetro para un command. Un ejemplo común es un parámetro para ser usado en un procedimiento almacenado.
- **DataAdapter:** "Puente" utilizado para transferir data entre una fuente de datos y un objeto DataSet (ver abajo).
- **DataReader:** Es una clase usada para procesar eficientemente una lista grande de resultados, un registro a la vez.

DataSets

Los objetos DataSets, son un grupo de clases que describen una simple base de datos relacional en memoria, fueron la estrella del show en el lanzamiento inicial (1.0) del Microsoft .NET Framework. Las clases forman una jerarquía de contención:

- Un objeto DataSet representa un esquema (o una base de datos entera o un subconjunto de una). Puede contener las tablas y las relaciones entre esas tablas.
  - Un objeto DataTable representa una sola tabla en la base de datos. Tiene un nombre, filas, y columnas.
    - Un objeto DataView "se sienta sobre" un DataTable y ordena los datos (como una cláusula "order by" de SQL) y, si se activa un filtro, filtra los registros (como una cláusula "where" del SQL). Para facilitar estas operaciones se usa un índice en memoria. Todas las DataTables tienen un filtro por defecto, mientras que pueden ser definidos cualquier número de DataViews adicionales, reduciendo la interacción con la base de datos subyacente y mejorando así el desempeño.
      - Un DataColumn representa una columna de la tabla, incluyendo su nombre y tipo.
      - Un objeto DataRow representa una sola fila en la tabla, y permite leer y actualizar los valores en esa fila, así como la recuperación de cualquier fila que esté relacionada con ella a través de una relación de clave primaria - clave extranjera.
      - Un DataRowView representa una sola fila de un DataView, la diferencia entre un DataRow y el DataRowView es importante cuando se está interactuando sobre un resultset.
  - Un DataRelation es una relación entre las tablas, tales como una relación de clave primaria - clave ajena. Esto es útil para permitir la funcionalidad del DataRow de recuperar filas relacionadas.
  - Un Constraint describe una propiedad de la base de datos que se debe cumplir, como que los valores en una columna de clave primaria deben ser únicos. A medida que los datos son modificados cualquier violación que se presente causará excepciones.

Un DataSet es llenado desde una base de datos por un DataAdapter cuyas propiedades Connection y Command que han sido iniciados. Sin embargo, un DataSet puede guardar su contenido a XML (opcionalmente con un esquema XSD), o llenarse a sí mismo desde un XML, haciendo esto excepcionalmente útil para los servicios web, computación distribuida, y aplicaciones ocasionalmente conectadas desconectados.

### **¿Qué es la “Inyección SQL”? ¿Cómo se puede evitar con ADO.NET?**

La inyección SQL es un método de infiltración de código intruso que se vale de una vulnerabilidad informática presente en una aplicación en el nivel de validación de las entradas para realizar operaciones sobre una base de datos.

El origen de la vulnerabilidad radica en la incorrecta comprobación o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL. Es, de hecho, un error de una clase más general de vulnerabilidades que puede ocurrir en cualquier lenguaje de programación o script que esté incrustado en otro.

Se conoce como Inyección SQL, indistintamente, al tipo de vulnerabilidad, al método de infiltración, al hecho de incrustar código SQL intruso y a la porción de código incrustado.

Se dice que existe o se produjo una inyección SQL cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos.

Este tipo de intrusión normalmente es de carácter malicioso, dañino o espía, por tanto es un problema de seguridad informática, y debe ser tomado en cuenta por el programador de la aplicación para poder prevenirlo. Un programa elaborado con descuido, displicencia o con ignorancia del problema, podrá resultar ser vulnerable, y la seguridad del sistema (base de datos) podrá quedar eventualmente comprometida.

La intrusión ocurre durante la ejecución del programa vulnerable, ya sea, en computadores de escritorio o bien en sitios Web, en este último caso obviamente ejecutándose en el servidor que los aloja.

La vulnerabilidad se puede producir automáticamente cuando un programa "arma descuidadamente" una sentencia SQL en tiempo de ejecución, o bien durante la fase de desarrollo, cuando el programador explicita la sentencia SQL a ejecutar en forma desprotegida. En cualquier caso, siempre que el programador necesite y haga uso de parámetros a ingresar por parte del usuario, a efectos de consultar una base de datos; ya que, justamente, dentro de los parámetros es donde se puede incorporar el código SQL intruso.

Al ejecutarse la consulta en la base de datos, el código SQL inyectado también se ejecutará y podría hacer un sinnúmero de cosas, como insertar registros, modificar o eliminar datos, autorizar accesos e, incluso, ejecutar otro tipo de código malicioso en el computador.

### **¿Qué es un delegado? ¿Para qué sirve?**

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signature compatible y un tipo de valor devuelto. Puede invocar (o llamar) al método a través de la instancia del delegado. Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados. Cree un método personalizado y una clase, como un control de Windows, podrá llamar al método cuando se produzca un determinado evento. En el siguiente ejemplo se muestra una declaración de delegado:

```
public delegate int PerformCalculation(int x, int y);
```

Cualquier método de cualquier clase o struct accesible que coincida con el tipo de delegado se puede asignar al delegado. El método puede ser estático o de instancia. Esta flexibilidad significa que puede cambiar las llamadas de método mediante programación, o bien agregar código nuevo a las clases existentes. Esta capacidad de hacer referencia a un método como parámetro hace que los delegados sean idóneos para definir métodos de devolución de llamada. Puede escribir un método que compare dos objetos en la aplicación. Ese método se puede usar en un delegado para un algoritmo de ordenación. Como el código de comparación es independiente de la biblioteca, el método de ordenación puede ser más general. El código asociado a un delegado se invoca mediante un método virtual agregado a un tipo de delegado.

Los delegados tienen las propiedades siguientes:

- Los delegados son similares a los punteros de función de C++, pero los primeros están completamente orientados a objetos y, a diferencia de los punteros de C++ de funciones de miembro, los delegados encapsulan una instancia de objeto y un método.
- Los delegados permiten pasar los métodos como parámetros.
- Los delegados pueden usarse para definir métodos de devolución de llamada.
- Los delegados pueden encadenarse entre sí; por ejemplo, se puede llamar a varios métodos en un solo evento.
- No es necesario que los métodos coincidan exactamente con el tipo de delegado.

### **¿En qué se diferencian los delegados de los punteros a función vistos en el lenguaje C?**

Los delegados habilitan escenarios en los que otros lenguajes, como C++, Pascal y Modula, se han direccionado con punteros de función. A diferencia de los punteros de función de C++, sin embargo, los delegados están totalmente orientados a objetos y, a diferencia de los punteros de C++ a las funciones miembro, los delegados encapsulan una instancia de objeto y un método.

Una declaración de delegado define una clase que se deriva de la clase System.Delegate. Una instancia de delegado encapsula una lista de invocación, que es una lista de uno o varios métodos, a la que se hace referencia como una entidad a la que se puede llamar. En el caso de los métodos de instancia, una entidad a la que se puede llamar está formada por una instancia y un método en esa instancia. En el caso de los métodos estáticos, una entidad a la que se puede llamar consta simplemente de un método. Al invocar una instancia de delegado con un conjunto de argumentos adecuado, se invoca a cada una de las entidades a las que se puede llamar del delegado con el conjunto de argumentos especificado. Una propiedad interesante y útil de una instancia de delegado es que no sabe ni le interesan las clases de los métodos que encapsula; lo único que importa es que esos métodos sean compatibles (declaraciones de delegado) con el tipo de delegado. Esto hace que los delegados sean perfectamente adecuados para la invocación "anónima".

## **Clase 24 – Eventos:**

### **¿Qué es un evento?**

Un evento es un mensaje que envía un objeto cuando ocurre una acción. La acción podría deberse a la interacción del usuario, como hacer clic en un botón, o podría derivarse de cualquier otra lógica del programa, como el cambio del valor de una propiedad. El objeto que provoca el evento se conoce como emisor del evento. El emisor del evento no sabe qué objeto o método recibirá (controlará) los eventos que genera. El evento normalmente es un miembro del emisor del evento; por ejemplo, el evento Click es un miembro de la clase Button, y el evento PropertyChanged es un miembro de la clase que implementa la interfaz INotifyPropertyChanged.

Para definir un evento, se utiliza la palabra clave event de C# o Event de Visual Basic en la signature de la clase de eventos y se especifica el tipo de delegado para el evento. Los delegados se describen en la sección siguiente.

Normalmente, para generar un evento, se agrega un método marcado como protected y virtual (en C#) o Protected y Overridable (en Visual Basic). Asigne a este método el nombre OnEventName; por ejemplo, OnDataReceived. El método debe tomar un parámetro que especifica un objeto de datos de evento, que es un objeto de tipo EventArgs o un tipo derivado. Este método se proporciona para permitir que las clases derivadas reemplacen la lógica para generar el evento. Una clase derivada siempre debería llamar al método OnEventName de la clase base para asegurarse de que los delegados registrados reciben el evento.

### **¿Por qué los eventos son de un tipo delegado? ¿Cómo impacta esto a los posibles manejadores de ese evento?**

Un delegado es un tipo que tiene una referencia a un método. Un delegado se declara con una signature que muestra el tipo de valor devuelto y los parámetros para los métodos a los que hace referencia, y únicamente puede contener referencias a los métodos que coinciden con su signature. Por lo tanto, un delegado equivale a un puntero a función con seguridad o a una devolución de llamada. Una declaración de delegado es suficiente para definir una clase de delegado.

Los delegados tienen muchos usos en .NET. En el contexto de los eventos, un delegado es un intermediario (o un mecanismo de puntero) entre el origen del evento y el código que lo controla. Para asociar un delegado a un evento se incluye el tipo de delegado en la declaración del evento, como se muestra en el ejemplo de la sección anterior. Para obtener más información sobre los delegados, vea la clase Delegate.

.NET proporciona los delegados EventHandler y EventHandler<TEventArgs> que admiten la mayoría de los escenarios de eventos. Use el delegado EventHandler para todos los eventos que no incluyen datos de evento. Use el delegado EventHandler<TEventArgs> para los eventos que incluyen datos sobre el evento. Estos delegados no tienen ningún valor de tipo devuelto y toman dos parámetros (un objeto para el origen del evento y un objeto para los datos del evento).

Los delegados son de multidifusión, lo que significa que pueden guardar referencias a más de un método de control de eventos. Para obtener información detallada, vea la página de referencia de Delegate. Los delegados permiten realizar un control de eventos más flexible y detallado. Un delegado actúa como remitente de eventos de la clase que genera el evento y mantiene una lista de los controladores registrados para el evento.

Para los escenarios en que no funcionan los delegados EventHandler y EventHandler<TEventArgs>, puede definir un delegado. Los escenarios para los es necesario definir un delegado son poco habituales, como

cuando se debe ejecutar código que no reconoce genéricos. Los delegados se marcan con la palabra clave `delegate` de C# y `Delegate` de Visual Basic en la declaración. En el ejemplo siguiente se muestra cómo declarar un delegado denominado `ThresholdReachedEventHandler`.

## **Clase 25 – Métodos de extensión:**

### **¿Qué es un método de extensión? ¿Para qué sirve?**

Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original. Los métodos de extensión son métodos estáticos, pero se les llama como si fueran métodos de instancia en el tipo extendido. En el caso del código de cliente escrito en C#, F# y Visual Basic, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos definidos en un tipo.

Los métodos de extensión más comunes son los operadores de consulta LINQ estándar, que agregan funciones de consulta a los tipos `System.Collections.IEnumerable` y `System.Collections.Generic.IEnumerable<T>` existentes. Para usar los operadores de consulta estándar, inclúyalos primero en el ámbito con una directiva `using System.Linq`. A partir de ese momento, cualquier tipo que implemente `IEnumerable<T>` parecerá tener métodos de instancia como `GroupBy`, `OrderBy`, `Average`, etc. Puede ver estos métodos adicionales en la finalización de instrucciones de IntelliSense al escribir "punto" después de una instancia de un tipo `IEnumerable<T>`, como `List<T>` o `Array`.

Los métodos de extensión se definen como métodos estáticos, pero se les llama usando la sintaxis de método de instancia. Su primer parámetro especifica en qué tipo funciona el método. El parámetro va precedido del modificador `this`. Los métodos de extensión únicamente se encuentran dentro del ámbito cuando el espacio de nombres se importa explícitamente en el código fuente con una directiva `using`.

### **¿Qué características debe tener la clase que lo contiene?**

### **¿Qué características debe tener el método para ser considerado un método de extensión?**

Para definir un método de extensión debemos declarar una clase `static` y dentro de la misma declarar el método de extensión que también debe ser estático.