

# **Programa Académico Desarrollo de Software Nivel II**

**C# II**  
**Prof. Gabriel Roa**

**Tema I – Excepciones y Depuración**

## Contenido

<b>I – GESTIÓN DE EXCEPCIONES</b>	<b>2</b>
<b>I.1 – DEFINICIONES BÁSICAS</b>	<b>3</b>
<b>I.2 – MANEJO DE EXCEPCIONES</b>	<b>4</b>
I.2.1 – TRY	5
I.2.2 – CATCH	9
I.2.3 – FINALLY	16
<b>I.3 – EXCEPCIONES PROPIAS</b>	<b>18</b>
<b>I.4 – LOG DE EXCEPCIONES EN ARCHIVOS</b>	<b>25</b>
<b>II – DEPURACIÓN EN VISUAL STUDIO</b>	<b>28</b>
<b>II.1 – DEFINICIONES BÁSICAS</b>	<b>28</b>
<b>II.2 – ELEMENTOS DE DEPURACIÓN</b>	<b>29</b>
II.2.1 – BREAKPOINTS	35
II.2.2 – NAVEGACIÓN POR EL CÓDIGO	39
II.2.3 – VISOR DE OBJETOS	41
<b>III – REFERENCIAS BIBLIOGRÁFICAS</b>	<b>49</b>

## I – Gestión de Excepciones

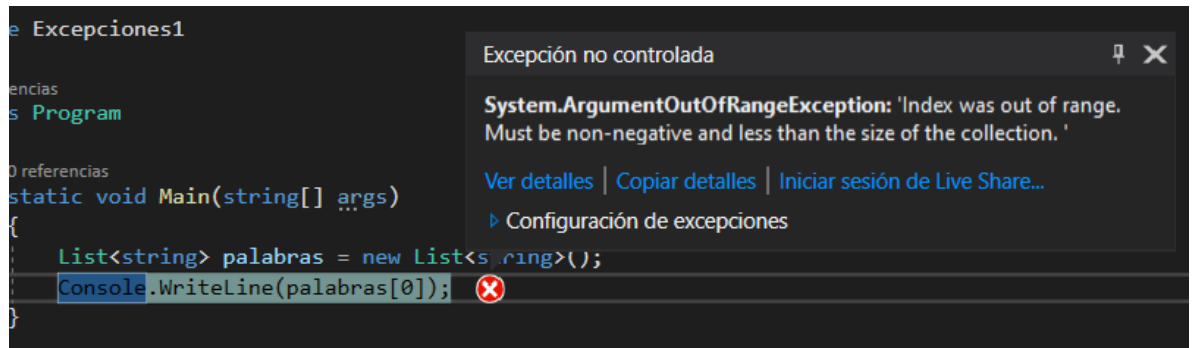
Hasta los momentos ha sido habitual que, en el momento que estemos programando, nos topemos con errores que finalizan de golpe la ejecución del programa que estamos realizando. Tomemos como ejemplo el siguiente fragmento de código:



```
static void Main(string[] args)
{
    List<string> palabras = new List<string>();
    Console.WriteLine(palabras[0]);
}
```

*Fig. 1 – Fragmento de código que arroja excepción*

Este fragmento de código está instanciando una List de string, declarándola e inicializándola como una List vacía, tras lo cual se intenta acceder al primer elemento del arreglo. ¿Qué sucede con este fragmento de código? Este no arrojará error alguno en tiempo de compilación, puesto que sintácticamente es válido; sin embargo, a la hora de ejecutar el programa, este arrojará una excepción, puesto que estoy intentando acceder al índice de un elemento inexistente. Basta ejecutar el programa para obtener este error:



*Fig. 2 – Excepción por acceder a un elemento inexistente.*

Este error es una excepción no controlada: un error excepcional que interrumpió el flujo del programa, que no fue detectado en tiempo de compilación, sino que sucedió en tiempo de ejecución. C#, al igual que básicamente todos los demás lenguajes de programación, provee una serie de herramientas, librerías y frameworks para gestionar las excepciones y evitar que estas interrumpan el flujo del programa.

### **I.1 – Definiciones básicas**

Una excepción es la indicación de que se produjo un error en el programa (Villalobos & Casallas). Al momento en que ocurre una excepción, la ejecución del programa se paraliza de manera excepcional – de allí su nombre, puesto que parten del hecho de que ocurrió una situación inesperada que provoca errores en el programa.

El tiempo de compilación es el momento en que un programa se compila de código fuente a lenguaje máquina. En este tiempo, se suele hacer chequeos

sintácticos básicos sobre el código, y si no hay ningún error sintáctico, el programa se compila para poderse ejecutar. Por otro lado, el tiempo de ejecución es el momento en que el programa está ejecutándose en el sistema, manipulando la memoria y procesando datos. Los errores que ocurren en tiempo de ejecución pueden generar excepciones o no, dependiendo del error en particular.

Una excepción puede ser controlada, esto es, una excepción que es manejada y atrapada por el programa, que efectuará una o más acciones dependiendo de lo establecido por el programador, o puede ser no controlada, en cuyo caso se paralizará y muy probablemente se finalizará la ejecución del programa. En todo caso, las excepciones son *lanzadas* (throw) por el programa, que buscará en el código si esta excepción es controlada (catch) o no, y actuará en consecuencia.

## **I.2 – Manejo de excepciones**

C# nos ofrece el manejo de excepciones a través de las cláusulas Try / Catch / Finally. Estas cláusulas nos permiten *intentar* ejecutar un código, *capturar* cualquier excepción que pueda ocurrir en su ejecución y *finalmente* ejecutar otro fragmento de código, independientemente de si sucede una excepción en tiempo de ejecución o no.

Su estructura es la siguiente:

A code editor window with a dark background and a purple border. It contains a Java try-catch-finally block. The code is as follows:

```
try {  
    //Código a intentar ejecutar  
}  
catch (Exception e)  
{  
    //Código a ejecutar si ocurre una excepción  
}  
finally  
{  
    //Código a ejecutar al finalizar el try y/o el catch  
}
```

*Fig. 3 – Estructura de una cláusula try – catch – finally*

### **I.2.1 – Try**

Dentro del try, va especificado todo el código que puede provocar una excepción. Es una buena práctica de programación detenerse a considerar: *¿el código que estoy escribiendo puede arrojar una excepción?* Y, dependiendo de la respuesta, entender si es necesario o no envolver nuestra expresión en un bloque try. Por ejemplo, consideremos el siguiente método:

```
static void Main(string[] args)
{
    double n1, n2, res;
    n1 = 24;
    n2 = 421.21;
    res = n1 + n2;
    Console.WriteLine(res);
}
```

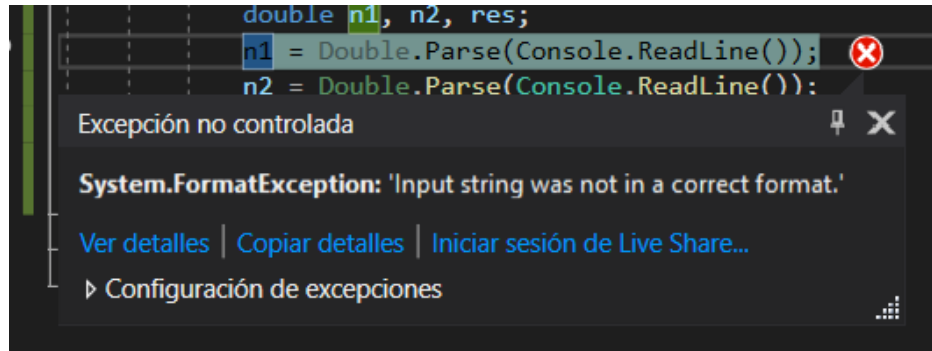
*Fig. 4 – Código que no arroja excepciones*

¿Este código podría arrojar alguna excepción? Considerando que estamos sumando dos números reales previamente establecidos, no hay ningún margen de error para que surja algún tipo de excepción. Ahora bien, es distinto si hacemos lo siguiente:

```
static void Main(string[] args)
{
    double n1, n2, res;
    n1 = Double.Parse(Console.ReadLine());
    n2 = Double.Parse(Console.ReadLine());
    res = n1 + n2;
    Console.WriteLine(res);
}
```

*Fig. 5 – Calculadora que suma dos números leídos por consola, código propenso a excepciones*

En este caso, el código sí tiene propensión a arrojar excepciones, puesto que se están recibiendo datos de entrada del teclado. ¿Qué sucede si, por ejemplo, introducimos un texto blanco?



*Fig. 6 – Excepción por intentar procesar una línea que no contenga un número válido.*

El código sí arroja una excepción. Por eso, en este caso es necesario envolverlo en un bloque try – catch.

```
static void Main(string[] args)  
{  
    double n1, n2, res;  
    try  
    {  
        n1 = Double.Parse(Console.ReadLine());  
        n2 = Double.Parse(Console.ReadLine());  
        res = n1 + n2;  
        Console.WriteLine(res);  
    }  
}
```

*Fig. 7 – Cláusula try en el código anterior.*



Sin embargo, un try no puede escribirse sin un catch que detecte las excepciones arrojadas en el mismo. Es por ello que la estructura resultante debe ser la siguiente:

A screenshot of a code editor with a dark background and a purple border. The code is in C# and demonstrates a try-catch block. The try block contains three lines of code: parsing two input strings into doubles and adding them. The catch block is empty, with a comment indicating where exception handling code would go.

```
static void Main(string[] args)
{
    double n1, n2, res;
    try
    {
        n1 = Double.Parse(Console.ReadLine());
        n2 = Double.Parse(Console.ReadLine());
        res = n1 + n2;
        Console.WriteLine(res);
    } catch (Exception e)
    {
        //Código a ejecutar cuando suceda una excepción
    }
}
```

*Fig. 8 – Try – catch en el ejemplo descrito.*

En este caso, también debemos envolver la suma y la impresión en consola de los números dentro del bloque try, puesto que, si falla la lectura de alguno de los números, no se podrá realizar la sumatoria.

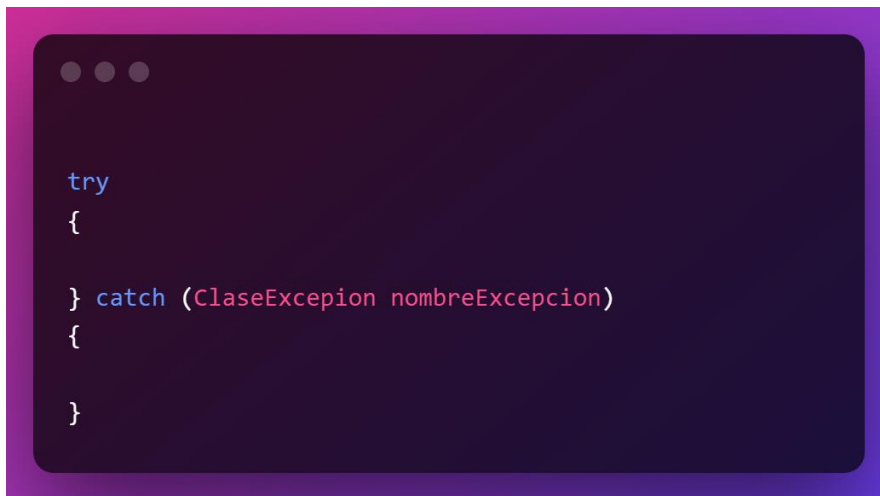
Ahora podemos describir la cláusula catch.

### I.2.2 – Catch

La cláusula Catch indica el código que se va a ejecutar en caso de que se detecte una excepción. Habitualmente, el código del catch realiza una de las siguientes actividades:

- ➔ Registrar la información de la excepción, estado del sistema y estado del equipo en un log de errores, que es un historial en que se encuentran todas las excepciones sucedidas hasta el momento.
- ➔ Notificar al usuario que hubo algún fallo en la ejecución del sistema.
- ➔ Reintentar la acción que arrojó error.
- ➔ Intentar otra acción distinta.

En cualquier caso, la estructura del catch es similar a la siguiente:

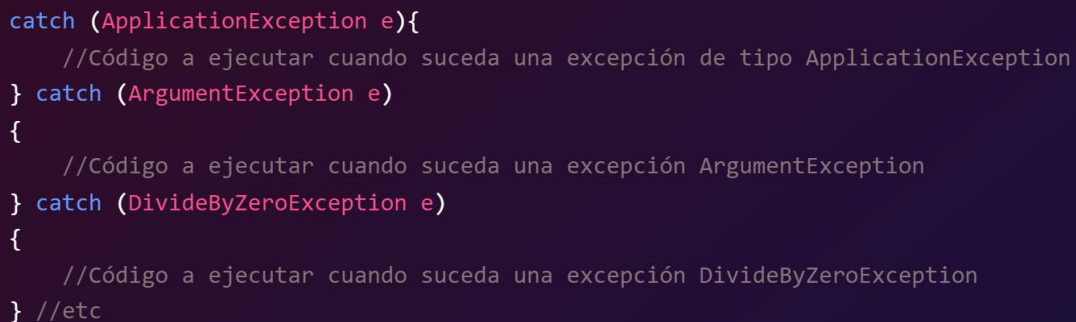
Un diagrama que representa un editor de código con un fondo oscuro y una barra de título púrpura. En el interior, se muestra un fragmento de código en Java con una estructura de try-catch. El código está escrito en un color claro (blanco o gris claro) sobre el fondo oscuro. El código es: try { } catch (ClaseExcepcion nombreExcepcion) { } }.

```
try
{
} catch (ClaseExcepcion nombreExcepcion)
{
}
}
```

*Fig. 9 – Estructura del catch*

En C#, toda excepción arrojada se corresponde a un objeto. Estos objetos pertenecen a una clase que describe el comportamiento, los datos e información de la excepción que se arrojó. Existen infinidad de clases de excepciones, y la principal característica que deben tener es que todas heredan de la clase genérica `Exception`.

A la hora de establecer un `catch`, lo ideal es que tengamos un `catch` que pueda captar cada uno de los tipos de excepción posibles que puedan arrojarse en la ejecución del `try`, considerando que es perfectamente válido realizar un anidamiento de `catch` en el que cada uno de los `catch` pueda atrapar una excepción distinta. Así, por ejemplo, esto es posible:



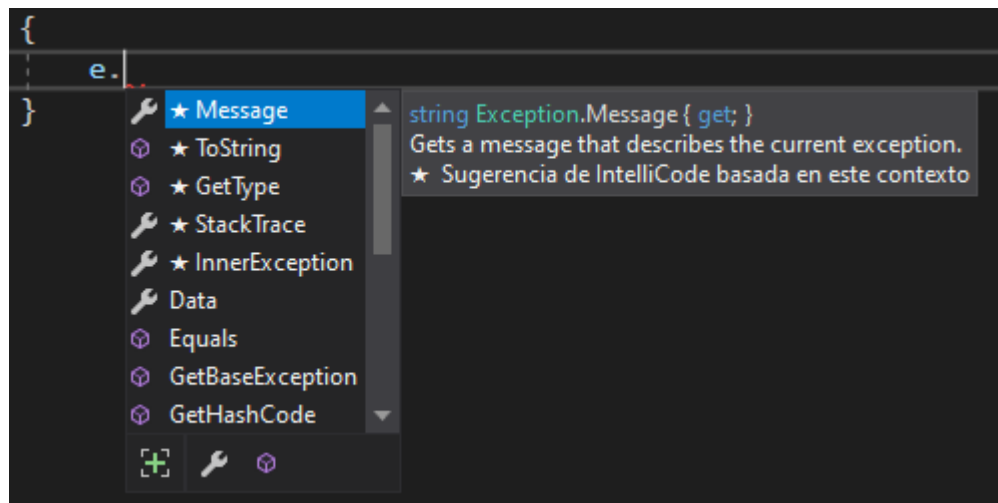
```
catch (ApplicationException e){  
    //Código a ejecutar cuando suceda una excepción de tipo ApplicationException  
} catch (ArgumentException e)  
{  
    //Código a ejecutar cuando suceda una excepción ArgumentException  
} catch (DivideByZeroException e)  
{  
    //Código a ejecutar cuando suceda una excepción DivideByZeroException  
} //etc
```

*Fig. 11 – Anidamiento de catch*

En este caso, cada `catch` se ejecutará sólo cuando se detecte una excepción de ese tipo en particular.

Del mismo modo, se puede captar una excepción genérica utilizando el código de la *figura 8*, a través del cual podemos notar que se capta la excepción de la clase `Exception`. Dado que todas las clases heredan de la clase `Exception`, esta cláusula `catch (Exception e)` permitirá capturar todas las excepciones que surjan, indistintamente de su tipo.

Ahora bien, podemos notar que tenemos una `e` al lado del nombre de la clase. Esta `e` se corresponde al nombre del objeto que guardará la información de la excepción, lo que nos permite acceder a ella y todos los atributos y métodos que mantiene.



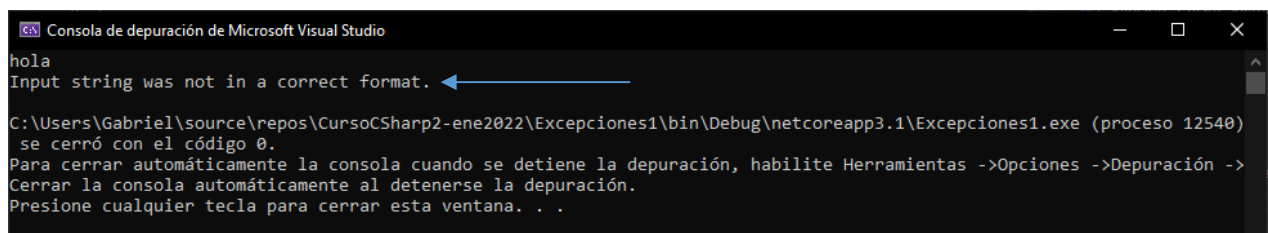
*Fig. 12 – Algunas de las propiedades y métodos de la clase `Exception`*

Ahora, volviendo al caso anterior, escribamos un catch que detecte una excepción e imprima el mensaje de la misma:

```
try
{
    n1 = Double.Parse(Console.ReadLine());
    n2 = Double.Parse(Console.ReadLine());
    res = n1 + n2;
    Console.WriteLine(res);
} catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

*Fig. 13 – Imprimiendo en consola una excepción*

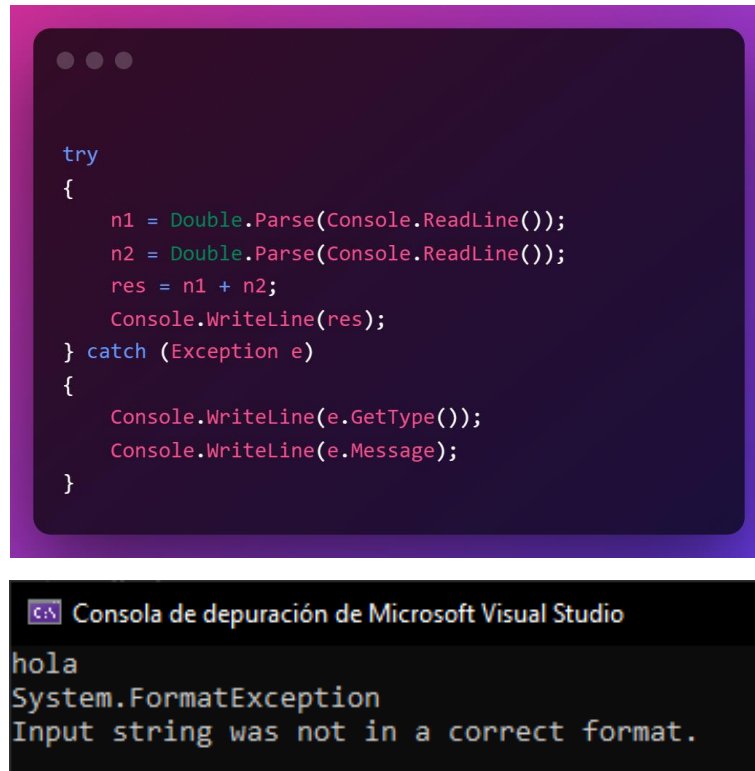
Sin mucha complicación, el programa sólo imprime en consola el mensaje de la excepción. Si ejecutamos y provocamos una excepción al introducir una palabra en uno de los datos de entrada, tendremos la siguiente salida:



hola  
Input string was not in a correct format. ←  
C:\Users\Gabriel\source\repos\CursoCSharp2-ene2022\Excepciones1\bin\Debug\netcoreapp3.1\Excepciones1.exe (proceso 12540) se cerró con el código 0.  
Para cerrar automáticamente la consola cuando se detiene la depuración, habilite Herramientas ->Opciones ->Depuración -> Cerrar la consola automáticamente al detenerse la depuración.  
Presione cualquier tecla para cerrar esta ventana. . .

*Fig. 14 – Excepción impresa en pantalla*

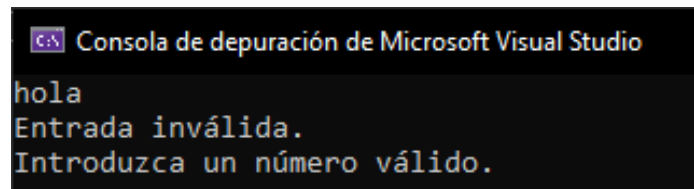
Sin embargo, esto no es lo ideal, puesto que estamos trabajando con la clase genérica `Exception`. En su lugar, debemos identificar cuál es la clase de la excepción que está sucediendo en ese momento en particular. Podemos apoyarnos del método `GetType()`:



*Fig. 15 – Identificando la excepción*

Ya que sabemos que se trata de una `FormatException`, podemos reemplazar `Exception` por `FormatException` para así poder escribir un `catch` adecuado para esta excepción en particular:

```
try
{
    n1 = Double.Parse(Console.ReadLine());
    n2 = Double.Parse(Console.ReadLine());
    res = n1 + n2;
    Console.WriteLine(res);
} catch (FormatException e)
{
    Console.WriteLine("Entrada inválida.\nIntroduzca un número válido.");
}
```

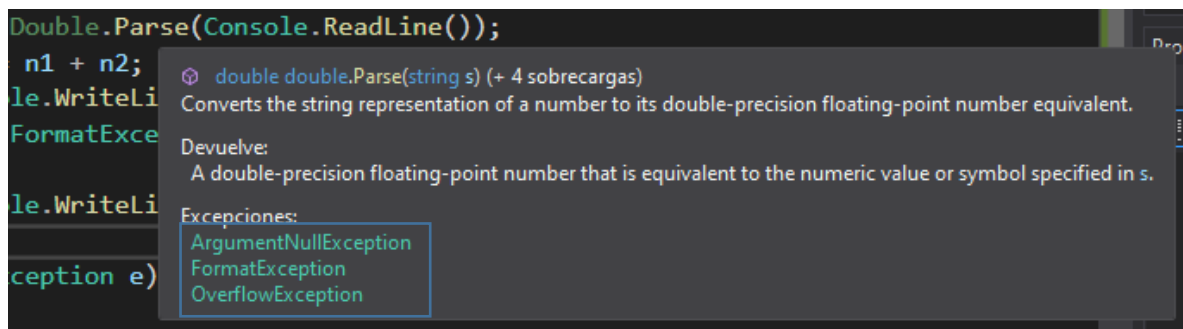


Consola de depuración de Microsoft Visual Studio

hola  
Entrada inválida.  
Introduzca un número válido.

Fig. 16 – Capturando un `FormatException`

También podemos verificar en la información del método `Parse` cuáles son las posibles excepciones que puede arrojar, al dejar el puntero del ratón sobre el método:



`Double.Parse(Console.ReadLine());`  
`n1 + n2;`  
`le.Writeli`  
`FormatException`  
`le.Writeli`  
`ception e)`

**double double.Parse(string s) (+ 4 sobrecargas)**  
Converts the string representation of a number to its double-precision floating-point number equivalent.

**Devuelve:**  
A double-precision floating-point number that is equivalent to the numeric value or symbol specified in *s*.

**Excepciones:**  
ArgumentNullException  
FormatException  
OverflowException

Fig. 17 – Distintas excepciones que arroja `Double.Parse()`

Por lo que podemos escribir una cláusula catch para cada una de ellas.

A screenshot of a code editor with a dark background and a purple border. The code is written in C# and shows three catch blocks for exceptions thrown by Double.Parse(). The first catch block is for FormatException, the second for ArgumentNullException, and the third for OverflowException. Each block contains a Console.WriteLine statement with a message in Spanish.

```
catch (FormatException e)
{
    Console.WriteLine("Entrada inválida.\nIntroduzca un número válido.");
} catch (ArgumentNullException e)
{
    Console.WriteLine("Argumento nulo.");
} catch (OverflowException e)
{
    Console.WriteLine("Número demasiado grande.");
}
```

*Fig. 18 – Cubriendo todas las posibles excepciones de Double.Parse()*

Y aún así, pueden suceder otras excepciones dentro del código, excepciones que puede que no estemos al tanto que pueden suceder, por lo que para conseguir la máxima cobertura de las excepciones, también podemos escribir una última excepción genérica, lo que nos dejaría con el siguiente código:



```
try
{
    n1 = Double.Parse(Console.ReadLine());
    n2 = Double.Parse(Console.ReadLine());
    res = n1 + n2;
    Console.WriteLine(res);
} catch (FormatException e)
{
    Console.WriteLine("Entrada inválida.\nIntroduzca un número válido.");
} catch (ArgumentNullException e)
{
    Console.WriteLine("Argumento nulo.");
} catch (OverflowException e)
{
    Console.WriteLine("Número demasiado grande.");
} catch (Exception e)
{
    Console.WriteLine($"Ha ocurrido una excepción {e.GetType()} con mensaje {e.Message}.");
}
```

*Fig. 19 – Cobertura de todos los posibles casos de excepciones.*

Al final, debemos intentar tener la máxima cobertura posible de las excepciones, de tal manera que evitemos posibles errores que rompan con la ejecución de los programas de parte del usuario.

### **I.2.3 – Finally**

Este código es el que se ejecuta *finalmente*, tras la ejecución de bien sea el try o del catch. Habitualmente se puede utilizar para limpiar cualquier recurso, variable o actividad realizada dentro del try, o para notificar la finalización de la ejecución del bloque de código.

```
try
{
    n1 = Double.Parse(Console.ReadLine());
    n2 = Double.Parse(Console.ReadLine());
    res = n1 + n2;
    Console.WriteLine(res);
} catch (FormatException e)
{
    Console.WriteLine("Entrada inválida.\nIntroduzca un número válido.");
} catch (ArgumentNullException e)
{
    Console.WriteLine("Argumento nulo.");
} catch (OverflowException e)
{
    Console.WriteLine("Número demasiado grande.");
} catch (Exception e)
{
    Console.WriteLine($"Ha ocurrido una excepción {e.GetType()} con mensaje {e.Message}.");
} finally
{
    Console.WriteLine("¡Adiós!");
}
```

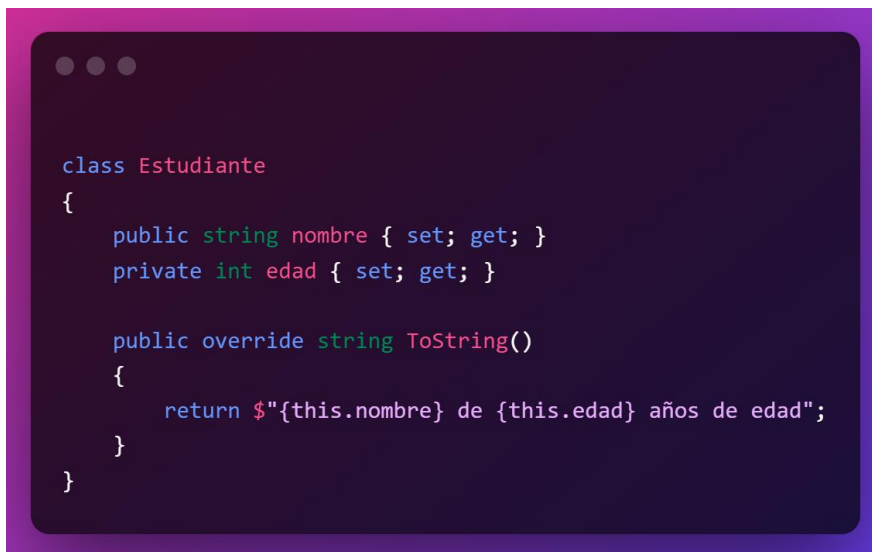
*Fig. 20 – Cláusula Finally*

Consola de depuración de Microsoft Visual Studio	Consola de depuración de Microsoft Visual Studio
10	hola
5	Entrada inválida.
15	Introduzca un número válido.
¡Adiós!	¡Adiós!

*Fig. 21 – Cláusula Finally en ejecución.*

### I.3 – Excepciones propias

C# ofrece la posibilidad de escribir métodos que puedan emitir excepciones, al igual que nos permite emitir una excepción en cualquier momento de la ejecución de un programa. Para ello, basta con utilizar la palabra reservada *throw*, acompañado de un objeto de una excepción que queramos emitir. Por ejemplo, consideremos un ejercicio en que tengamos una clase Estudiante:

The image shows a code editor window with a dark background and a purple border. Inside, the C# code for the 'Estudiante' class is displayed with syntax highlighting. The code defines a class with two properties, 'nombre' and 'edad', and a 'ToString()' method that returns a string representation of the student's information.

```
class Estudiante
{
    public string nombre { set; get; }
    private int edad { set; get; }

    public override string ToString()
    {
        return $"{this.nombre} de {this.edad} años de edad";
    }
}
```

*Fig. 22 – Clase Estudiante de ejemplo para emitir excepciones.*

Vamos a asumir que queremos escribir un código que valide que la edad del estudiante no pueda ser menor a 5 años. Para ello, haremos uso de las propiedades de C#: en principio, crearemos un atributo privado donde guardaremos la edad, y definiremos un set y un get personalizados para la propiedad edad.

```
class Estudiante
{
    public string nombre { set; get; }
    private int _edad;

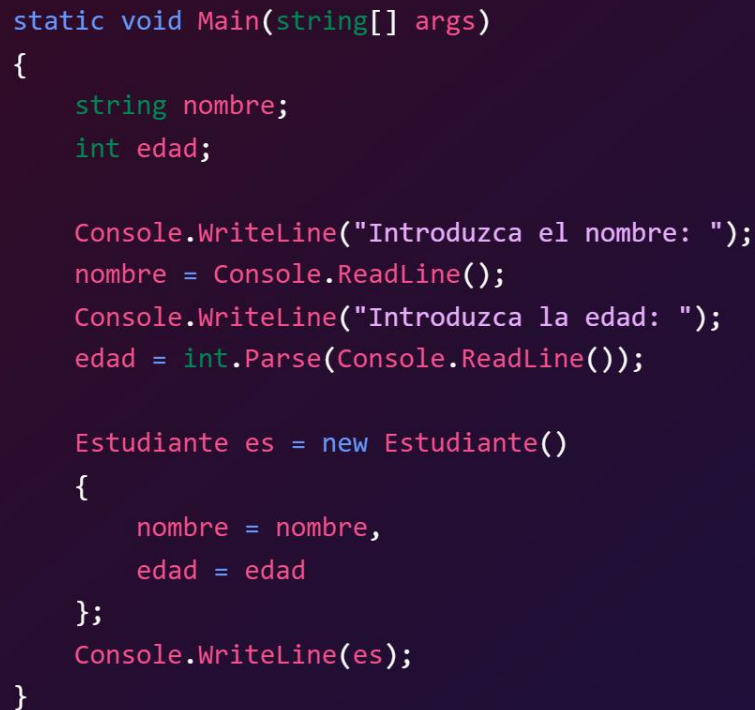
    public int edad {
        get { return _edad; }
        set {
            if (value < 5) throw new ArgumentOutOfRangeException("No puede ser menor de 5 años");
            _edad = value;
        }
    }

    public override string ToString()
    {
        return $"{this.nombre} de {this.edad} años de edad";
    }
}
```

*Fig. 23 – Emitiendo una excepción para validar la edad.*

Tenemos el atributo privado de edad, al mismo tiempo que tenemos la propiedad de edad que define el get y set para acceder a este. Si observamos con detenimiento el set, podemos ver que se realiza la validación: si el valor es menor a 5, emite una excepción. Como al momento de emitir una excepción debemos emitir un objeto que herede de Exception, vamos a crear una nueva instancia de una excepción. C# ofrece una gran cantidad de excepciones ya prediseñadas, y entre ellas, tenemos la `ArgumentOutOfRangeException`, que tal y como su nombre indica, es una excepción a emitir cuando el argumento que se está intentando definir está fuera de rango.

Consideremos ahora el siguiente código, para un programa que leerá nombre y edad de un estudiante y luego lo imprimirá en pantalla:



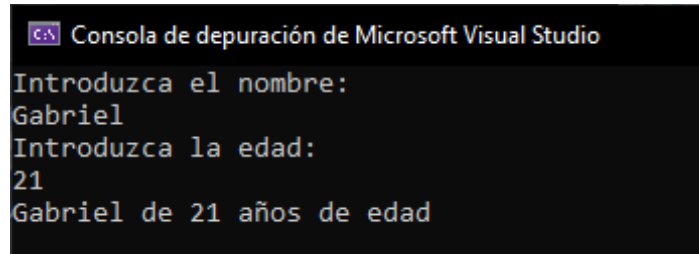
```
static void Main(string[] args)
{
    string nombre;
    int edad;

    Console.WriteLine("Introduzca el nombre: ");
    nombre = Console.ReadLine();
    Console.WriteLine("Introduzca la edad: ");
    edad = int.Parse(Console.ReadLine());

    Estudiante es = new Estudiante()
    {
        nombre = nombre,
        edad = edad
    };
    Console.WriteLine(es);
}
```

*Fig. 24 – Código para crear un estudiante y luego imprimirlo.*

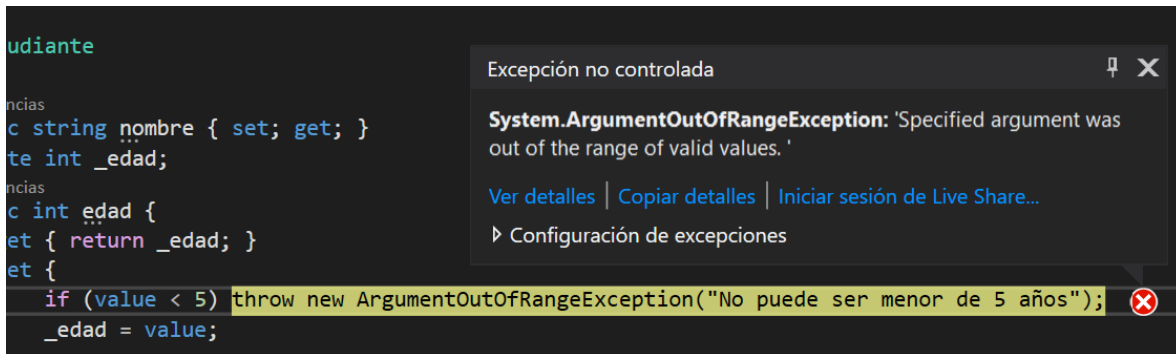
Cuando ejecutamos, vemos que el programa funciona sin mayor complicación.

A screenshot of the Visual Studio console window. The title bar reads "Consola de depuración de Microsoft Visual Studio". The console output shows the program's execution: "Introduzca el nombre:", "Gabriel", "Introduzca la edad:", "21", and "Gabriel de 21 años de edad".

```
Consola de depuración de Microsoft Visual Studio
Introduzca el nombre:
Gabriel
Introduzca la edad:
21
Gabriel de 21 años de edad
```

*Fig. 25 – Programa en ejecución.*

Sin embargo, si intentamos crear un estudiante de menos de 5 años de edad, obtenemos la siguiente excepción sin controlar:

A screenshot of the Visual Studio IDE. On the left, a C# code snippet for a student class is shown. The 'SetEdad' method contains a validation check: 'if (value < 5) throw new ArgumentOutOfRangeException("No puede ser menor de 5 años");'. On the right, an 'Unhandled Exception' dialog box is open, displaying the error: 'System.ArgumentOutOfRangeException: 'Specified argument was out of the range of valid values.''. It includes links for 'Ver detalles', 'Copiar detalles', and 'Iniciar sesión de Live Share...', and a 'Configuración de excepciones' button.

```
Estudiante
{
    string nombre { set; get; }
    int _edad;
    int edad {
        get { return _edad; }
        set {
            if (value < 5) throw new ArgumentOutOfRangeException("No puede ser menor de 5 años");
            _edad = value;
        }
    }
}
```

Excepción no controlada

**System.ArgumentOutOfRangeException:** 'Specified argument was out of the range of valid values.'

[Ver detalles](#) | [Copiar detalles](#) | [Iniciar sesión de Live Share...](#)

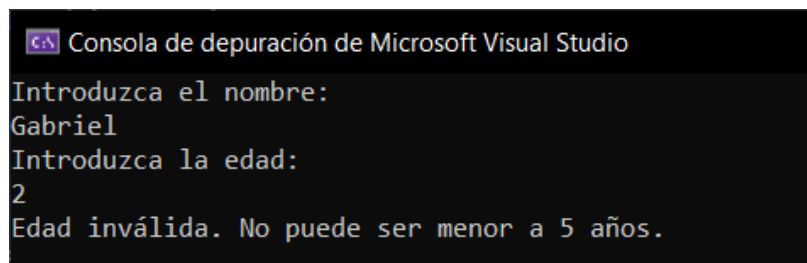
► Configuración de excepciones

*Fig. 26 – Excepción arrojada por el set edad del Estudiante.*

Evidentemente no queremos que esto corte la ejecución del programa, por lo que escribimos un bloque try – catch:

```
try
{
    Estudiante es = new Estudiante()
    {
        nombre = nombre,
        edad = edad
    };
    Console.WriteLine(es);
} catch (ArgumentOutOfRangeException)
{
    Console.WriteLine("Edad inválida. No puede ser menor a 5 años.");
}
```

*Fig. 27 – Bloque try – catch apropiado para este uso.*



```
Consola de depuración de Microsoft Visual Studio
Introduzca el nombre:
Gabriel
Introduzca la edad:
2
Edad inválida. No puede ser menor a 5 años.
```

*Fig. 28 – Excepción controlada.*

Ahora bien, también es posible que escribamos nuestra propia clase de excepción, en caso de que entre las excepciones de C#<sup>1</sup> no encontremos una

---

<sup>1</sup> Las excepciones más comunes pueden encontrarse en internet, entre la documentación del lenguaje.

que se adecúe al caso que queremos cubrir. Para ello, debemos escribir una clase que cumpla con las siguientes características:

- 1) Debe heredar de la clase Exception.
- 2) Su nombre debe finalizar con la palabra Exception.
- 3) Debe implementar los tres constructores de la clase Exception:
  - a. Exception(), que no recibe parámetro.
  - b. Exception(String), que recibe un mensaje.
  - c. Exception(String, Exception), que recibe un mensaje y otra excepción.

A sabiendas de ello, escribiremos una excepción personalizada para indicar que la edad está fuera de rango:



```
class EdadFueraDeRangoException : Exception
{
    public EdadFueraDeRangoException() { }

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

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

*Fig. 29 – Excepción personalizada para indicar que la edad está fuera de rango.*

Vemos que se cumplen todas las condiciones:



- 1) Hereda de la clase Exception.
- 2) Se llama EdadFueraDeRango**Exception**, por lo que su nombre finaliza con Exception.
- 3) Implementa los tres constructores, haciendo llamado a los constructores de la superclase.

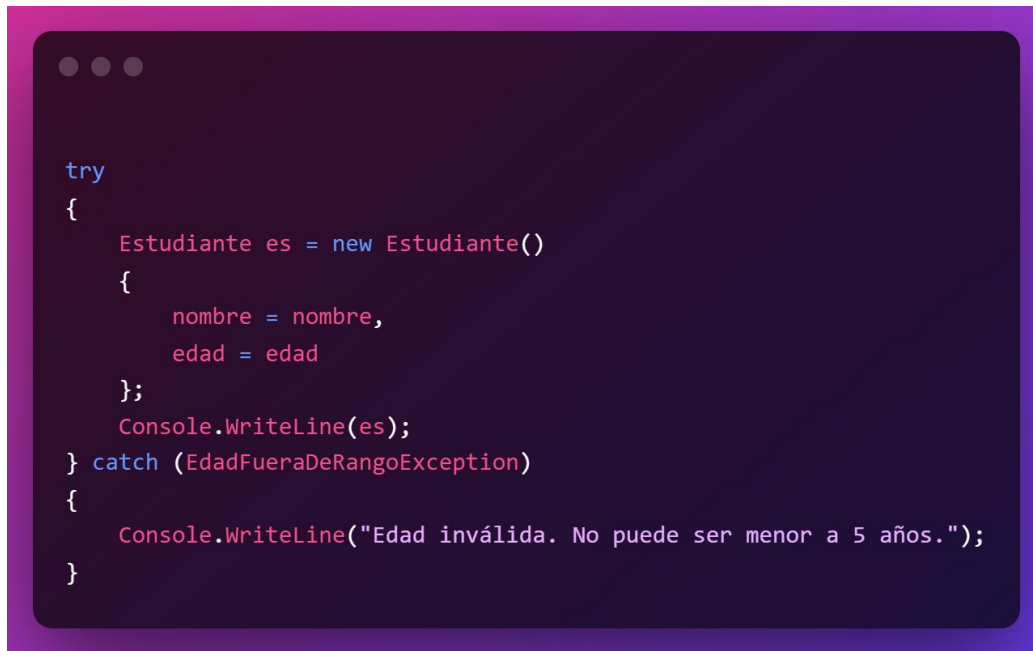
Así, podemos modificar el código de la propiedad edad:



```
public int edad {  
    get { return _edad; }  
    set {  
        if (value < 5) throw new EdadFueraDeRangoException("No puede ser menor de 5 años");  
        _edad = value;  
    }  
}
```

*Fig. 30 – Emitiendo la excepción personalizada.*

Y, del mismo modo, modificar la forma en que recibimos la excepción:



```
try
{
    Estudiante es = new Estudiante()
    {
        nombre = nombre,
        edad = edad
    };
    Console.WriteLine(es);
} catch (EdadFueraDeRangoException)
{
    Console.WriteLine("Edad inválida. No puede ser menor a 5 años.");
}
```

*Fig. 31 – Capturando la excepción personalizada.*

#### **I.4 – Log de excepciones en archivos**

Imprimir excepciones en consola puede funcionar para realizar una depuración pasajera, pero no para poder realizar un trabajo de depuración que dependa en mayor medida de la verificación de las excepciones y su estado actual. Para solucionar este problema, se ofrece con este curso una sencilla clase Logger, que realiza una implementación de un método LogError.

Este método recibe como parámetro una excepción, y guarda una descripción completa de la excepción ejecutada en el mismo directorio en que está ejecutándose el programa, además de devolver esta descripción como un string en caso de que desee imprimir en consola de igual manera.

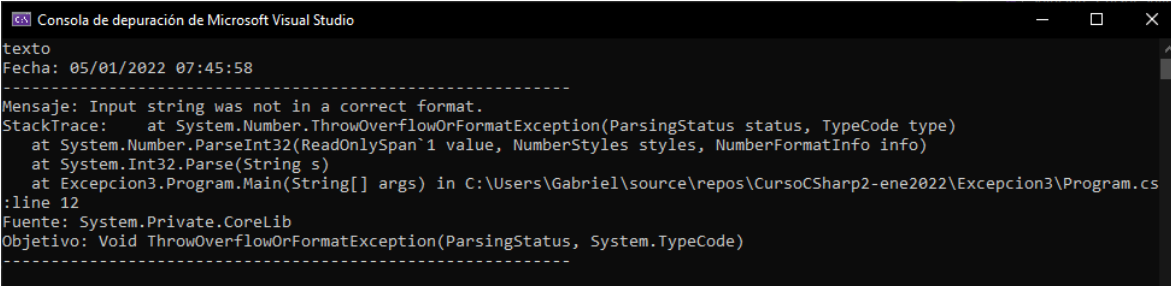
Así, tenemos la siguiente forma de loggear errores:



```
static void Main(string[] args)
{
    try
    {
        int x = int.Parse(Console.ReadLine());
    } catch (Exception e)
    {
        Console.WriteLine(Logger.LogError(e));
    }
}
```

*Fig. 32 – Usando Logger.LogError*

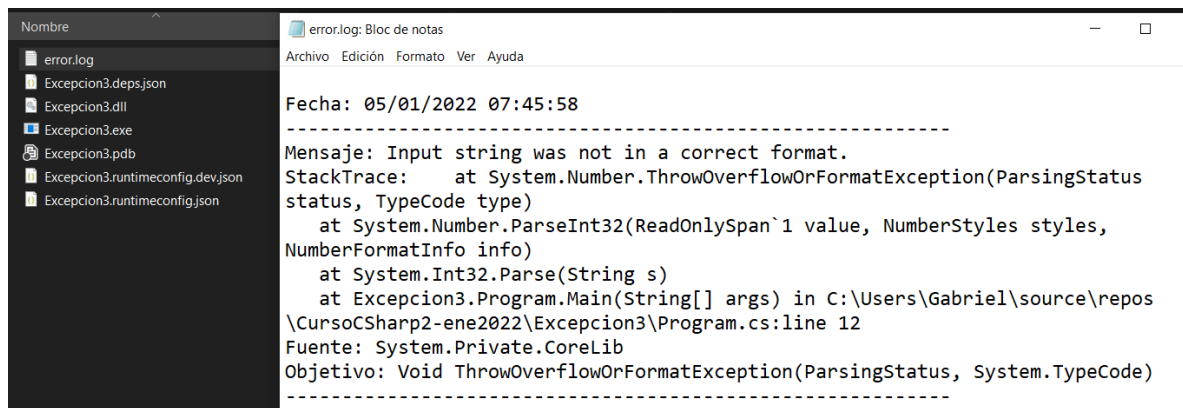
Al ejecutar el programa y forzar una excepción, tendremos la siguiente vista en consola:



```
Consola de depuración de Microsoft Visual Studio
texto
Fecha: 05/01/2022 07:45:58
-----
Mensaje: Input string was not in a correct format.
StackTrace:   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
             at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
             at System.Int32.Parse(String s)
             at Excepcion3.Program.Main(String[] args) in C:\Users\Gabriel\source\repos\CursoCSharp2-ene2022\Excepcion3\Program.cs:line 12
Fuente: System.Private.CoreLib
Objetivo: Void ThrowOverflowOrFormatException(ParsingStatus, System.TypeCode)
-----
```

*Fig. 33 – Error impreso en consola*


Y el mismo error en un archivo error.log ubicado en la carpeta donde se encuentra el ejecutable del programa.



*Fig. 34 – Archivo error.log*

## II – Depuración en Visual Studio

Una de las utilidades más importantes a la hora de utilizar un IDE es la de aprovechar las distintas herramientas de depuración que el mismo ofrece. Estas herramientas son las que nos permiten analizar el comportamiento de un programa en un momento dado y estudiar su funcionamiento, para verificar si es correcto o no. Más aún, en caso de encontrar cualquier tipo de error, nos brinda herramientas suficientes para poder identificar y solucionar los mismos.

Visual Studio no se queda atrás, y, de hecho, en el momento que ejecutamos un programa haciendo uso del botón , estaremos accediendo al modo de depuración, en el cual se ejecutará el programa con las herramientas de depuración activas.

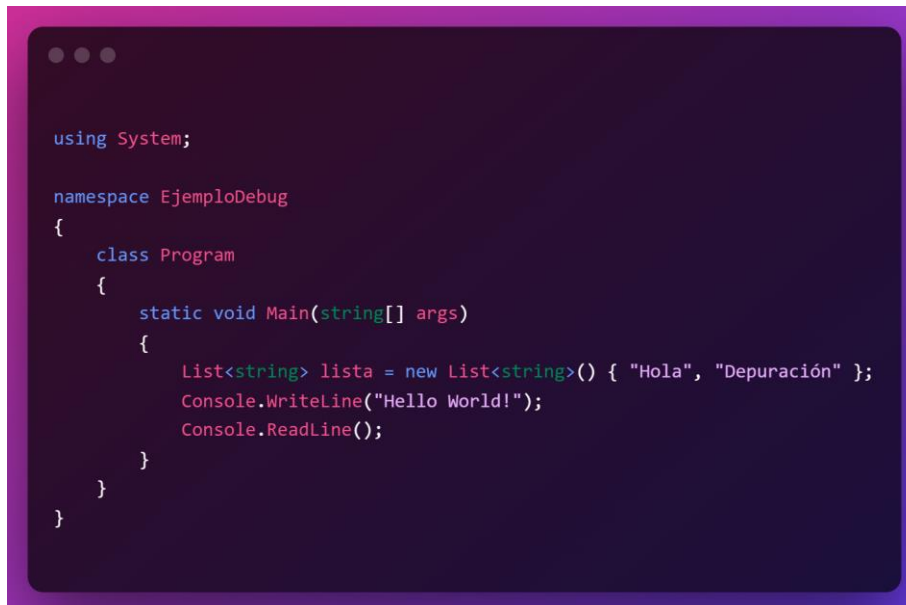
### II.1 – Definiciones básicas

La depuración es el procedimiento a través de cual se solucionan los errores que se encuentren en el código de un programa. Como es bien sabido, hay procedimientos de depuración llevados a cabo por el compilador, que nos indica cuando hay errores sintácticos en el código escrito; del mismo modo, hay procedimientos de análisis a través de los cuáles podemos identificar y solucionar los errores de lógica. Sin embargo, en su mayoría, se realiza es el proceso de depuración a través de un *depurador*.

Un depurador es un conjunto de herramientas que nos permiten llevar a cabo el proceso de depuración del programa. En su mayoría, son herramientas de desarrollo altamente especializadas que se pueden vincular a una aplicación que esté actualmente en ejecución para realizar una inspección cercana del código, cómo funciona y cuál es su estado en un momento dado (Microsoft, 2021).

## II.2 – Elementos de depuración

Para depurar errores sintácticos, Visual Studio nos ofrece pistas visuales que nos indican los errores actualmente en el código. Por ejemplo, consideremos el siguiente código:



```
using System;

namespace EjemploDebug
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> lista = new List<string>() { 'Hola', 'Depuración' };
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

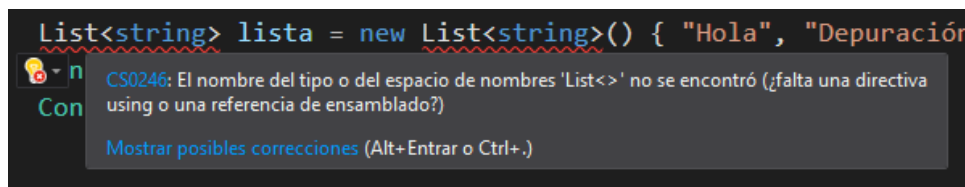
*Fig. 35 – Código con error sintáctico*

Visual Studio nos indicará, a través del subrayado, los errores que encuentre. Un subrayado en rojo indica que el fragmento de código tiene un error; un subrayado verde indica una advertencia. Si observamos el código anterior en Visual Studio, veremos lo siguiente:

```
0 referencias
static void Main(string[] args)
{
    List<string> lista = new List<string>() { "Hola", "Depuración" };
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

*Fig. 36 – Subrayado en rojo*

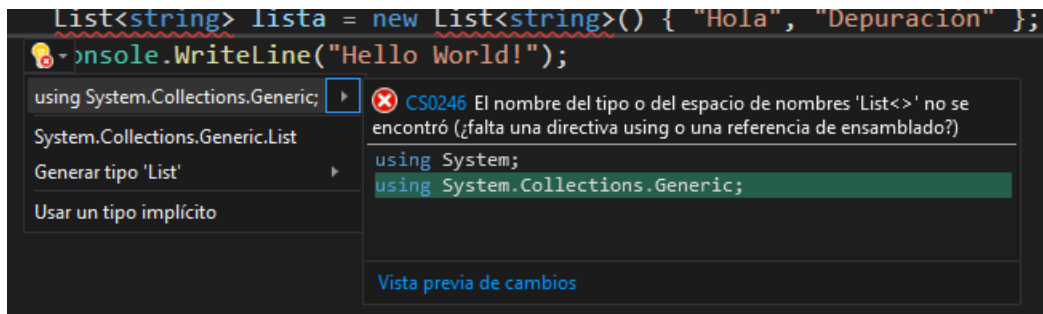
El subrayado en rojo nos indica un error en el programa. Si posamos el cursor del ratón en el texto subrayado, tendremos una descripción del error y un ícono de un bombillo.



*Fig. 37 – Descripción del error e ícono de bombillo.*

Ahora, leyendo el texto del recuadro podemos entender que no se ha definido `List<>`, lo que muy probablemente implica que no hemos hecho la importación de `System.Collections.Generic`. Podríamos volver al código y

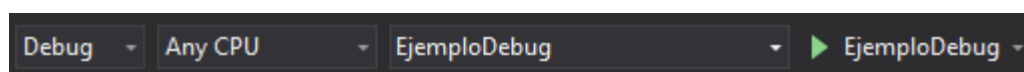
anexar la importación, o bien podríamos dar click al bombillo, que desplegará las posibles correcciones del error:



*Fig. 38 – Posibles correcciones del error.*

Al hacer click en esta opción, tendremos desplegado un menú contextual a través del cual se nos mostrarán distintas soluciones que podemos aplicar para nuestro error. Como en este caso estamos seguros de que sólo faltó realizar la importación de `System.Collections.Generic`, entonces sólo basta con hacer click en esta primera opción para que el IDE realice la importación de manera automática. Al hacerlo, los errores desaparecerán.

Ahora bien, si no se trata de errores sintácticos sino de errores de lógica, acceder al modo de depuración del programa es tan sencillo como ejecutar teniendo seleccionada la opción de Debug.



*Fig. 39 – Ejecutar en modo de depuración*



Ejecutar el programa con Debug seleccionado permitirá que utilicemos la depuración de Visual Studio en él, ejecutando el programa en modo de depuración; mientras que el programa se ejecuta, Visual Studio mantendrá una pantalla similar a la siguiente:

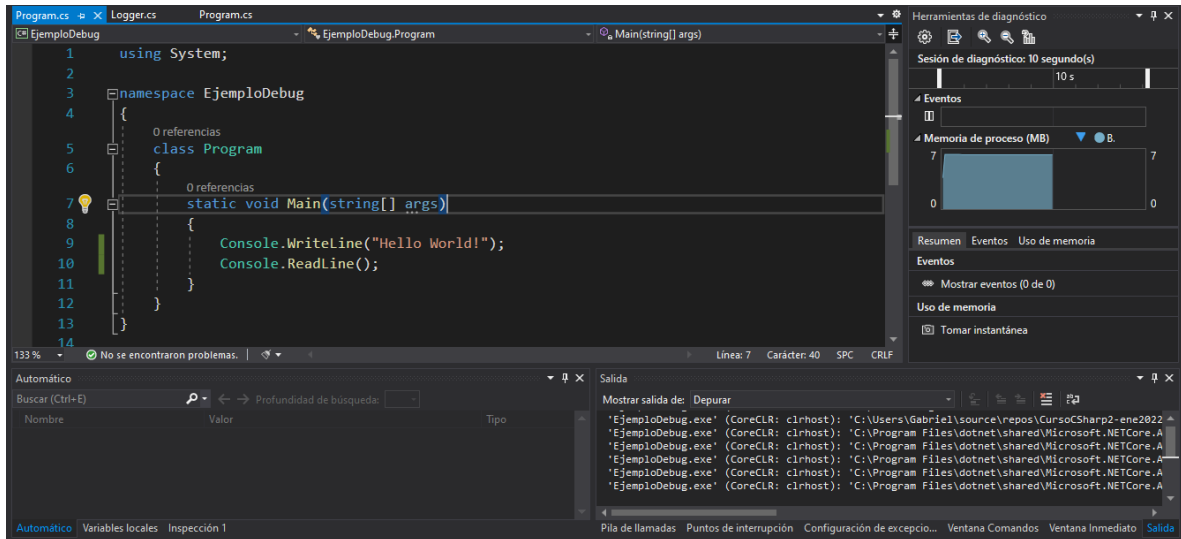


Fig. 40 – Depuración en Visual Studio

En esta pantalla tendremos:

- 1) El inspector de objetos, que nos mostrará las variables locales, sus valores y sus tipos.

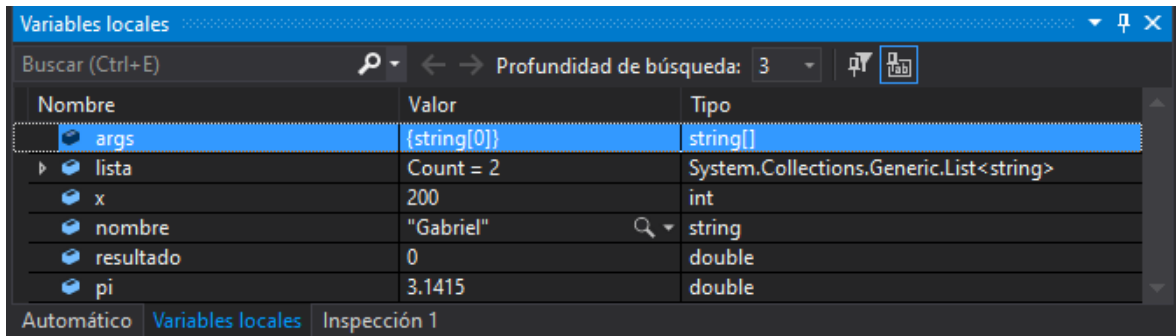


Fig. 41 – Inspector de objetos.

- 2) La consola de salida del programa, que mostrará los resultados de depuración, compilación, etc, y en caso de programas de WinForms, funcionará como la consola a la que imprimiremos datos a través de `Console.WriteLine`.

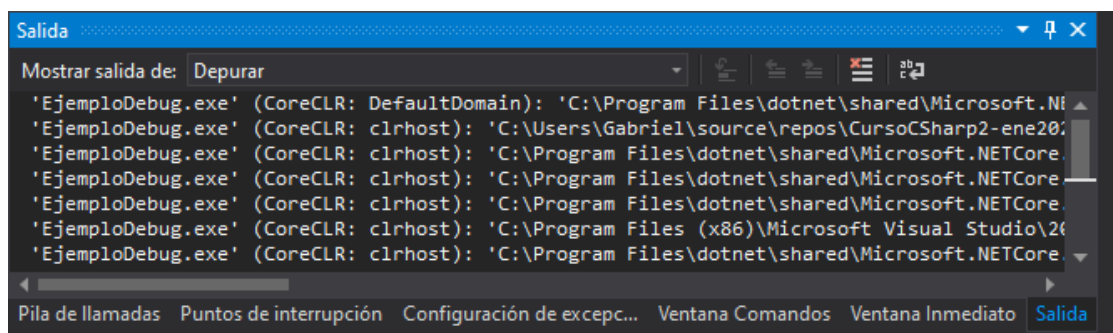


Fig. 42 – Consola de Salida

Del mismo modo, en esta opción también podremos ver los breakpoints activos en el programa.

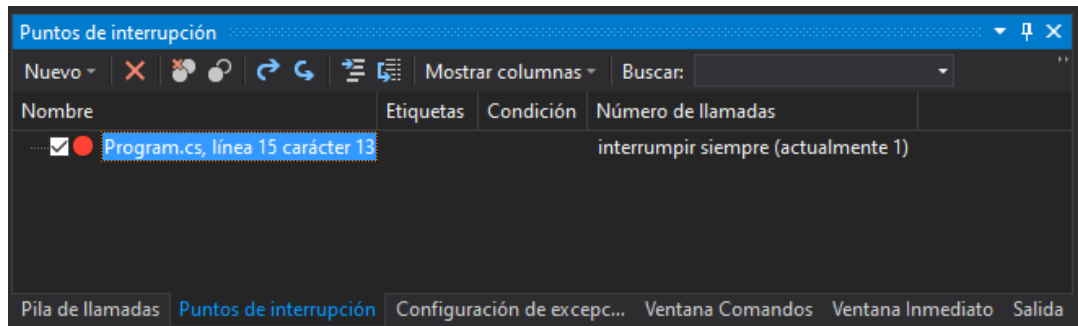


Fig. 43 – Visor de Breakpoints o Puntos de interrupción

- 3) Las herramientas de diagnóstico que mostrarán el rendimiento actual del programa, su uso de memoria, etc.

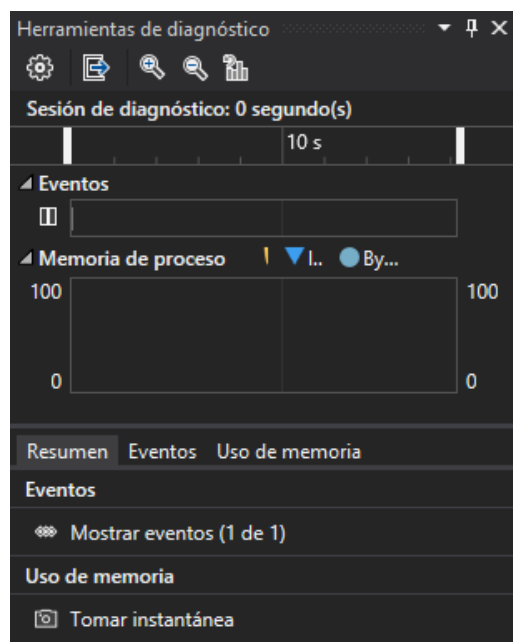


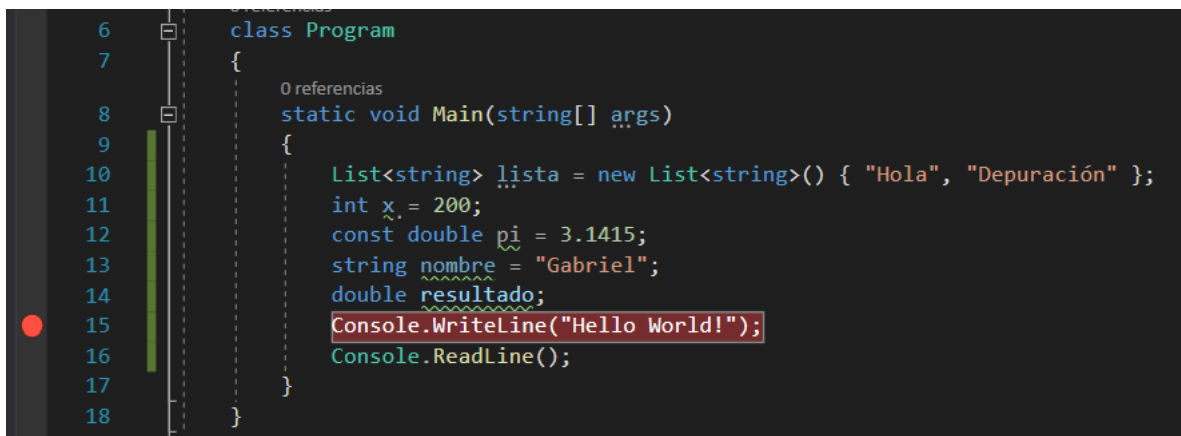
Fig. 44 – Herramientas de diagnóstico

Para hacer uso de todas estas herramientas, deberemos crear Breakpoints o Puntos de Interrupción que nos permitan paralizar la ejecución del programa.

## II.2.1 – Breakpoints

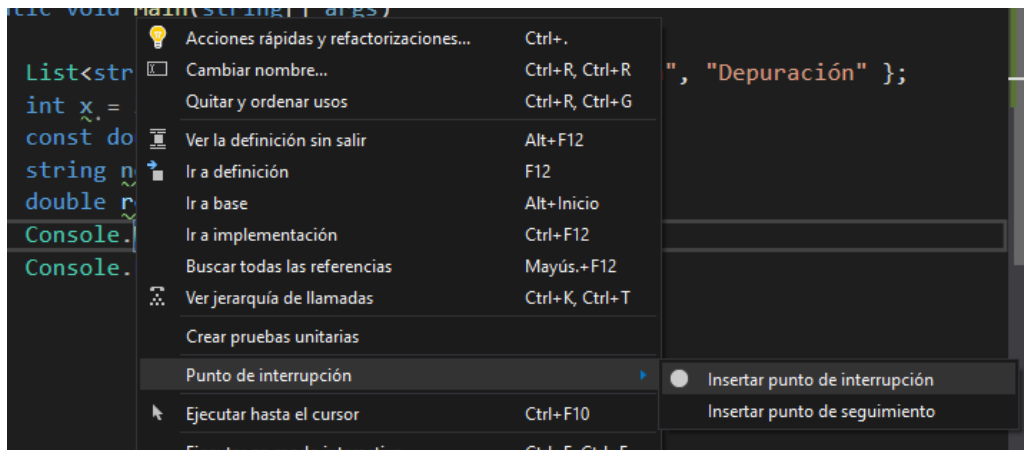
Conocidos en español como Puntos de Interrupción, son unos puntos a partir de los cuales la ejecución del programa en modo de depuración se paralizará, dando paso a la ejecución de las herramientas de depuración.

Hay distintas maneras de crear breakpoints. La primera y más habitual es la de hacer click en la barra izquierda, justo en la línea de código donde queramos detener el programa. Al hacerlo, el código se subrayará de color rojo, lo que indicará que en esa línea de código habrá un breakpoint.



*Fig. 45 – Breakpoint en la línea 15. Se debe hacer click en donde está situado el círculo rojo.*

Eliminar un breakpoint es tan sencillo como volver a hacer click en este círculo rojo. Otra manera de crear un breakpoint es haciendo click derecho en la línea que queramos introducir el breakpoint – Puntos de interrupción – Insertar puntos de interrupción:



*Fig. 46 – Otra manera de crear un breakpoint*

Consideremos el siguiente fragmento de código:

```

static void Main(string[] args)
{
    const double PI = 3.14159;
    List<string> lista = new List<string>() { "Hola", "Depuración" };
    int radio = 200;
    string nombre = "Circunferencia";
    double area;

    area = PI * Math.Pow(radio, 2);

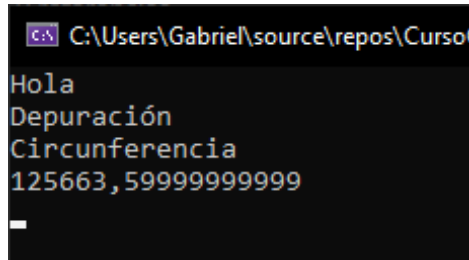
    lista.Add(nombre);
    lista.Add(area.ToString());

    foreach (string elemento in lista)
    {
        Console.WriteLine(elemento);
    }

    Console.ReadLine();
}
  
```

*Fig. 47 – Fragmento de código a depurar*

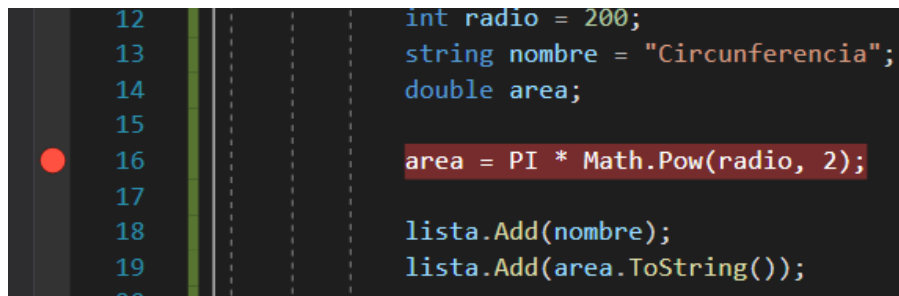
Este código calcula el área de una circunferencia y luego la agrega a una `List<string>`, junto a la palabra circunferencia, para finalmente imprimir todos los elementos de la misma. Si ejecutamos, tendremos la siguiente salida por consola:



```
C:\Users\Gabriel\source\repos\Curso0
Hola
Depuración
Circunferencia
125663,599999999999
_
```

*Fig. 48 – Programa en ejecución*

Ahora, ejecutaremos en modo debug introduciendo un breakpoint en la siguiente línea de código:



```
12 int radio = 200;
13 string nombre = "Circunferencia";
14 double area;
15
16 area = PI * Math.Pow(radio, 2);
17
18 lista.Add(nombre);
19 lista.Add(area.ToString());
20
```

*Fig. 49 – Breakpoint en línea 16*

Cuando ejecutamos, el programa se detendrá momentáneamente cuando llegue a la línea 16, mostrando la siguiente pantalla en Visual Studio:

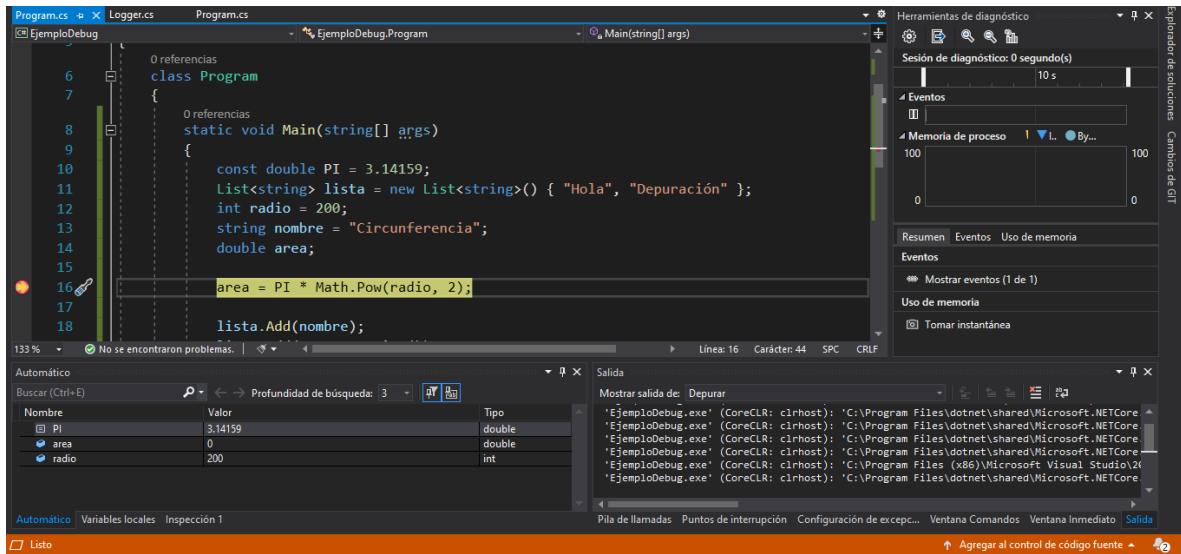


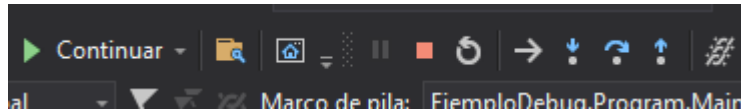
Fig. 50 – Breakpoint en ejecución

Al suceder esta interrupción, podremos hacer uso de todas las herramientas de depuración, navegar por el código ejecutando línea a línea o función a función, utilizar el visor de objetos, entre muchas otras opciones de depuración.

Podemos establecer cualquier cantidad de breakpoints que queramos, en cualquier línea de código que deseemos. Así, podemos tener un breakpoint que se ejecute con el constructor de un objeto, que se ejecute junto al llamado de un método, que se ejecute con una asignación, básicamente cualquier línea de código que no esté vacía puede tener un breakpoint. Del mismo modo, una vez que el programa detiene su ejecución por entrar en un breakpoint, podremos crear más y que el programa se detenga en ellos.

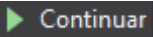
## II.2.2 – Navegación por el código

Una vez que un breakpoint está en ejecución, podremos notar las siguientes características en Visual Studio:




*Fig. 51 – Nuevos botones para navegar por el código*

Estos botones nos permiten navegar por el código y su depuración de la siguiente manera:

 **Continuar** Continúa con la ejecución del programa hasta el próximo breakpoint.


 Detiene la ejecución del programa.

 Reinicia el programa.

 Sitúa el foco sobre la próxima línea de código a ejecutarse.

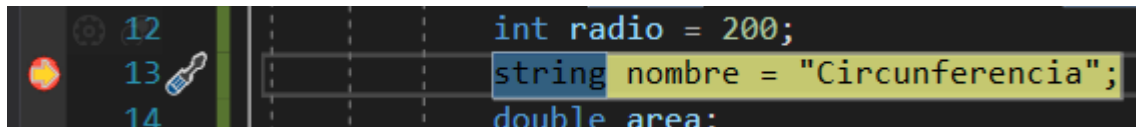
 Ejecuta la siguiente línea de código

 Ejecuta la siguiente línea de código, *saltando* las funciones y métodos.

 Ejecuta la siguiente función, *saltando* la función actual.

Asimismo, tenemos también una flecha de color amarillo, situada en la misma barra donde se encuentran los breakpoints, a través de la cual podemos seleccionar a discreción cuál será la siguiente línea de código en ejecutarse:

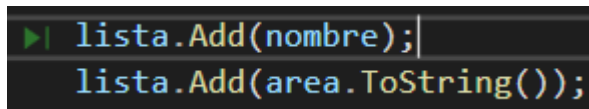




*Fig. 52 – Flecha amarilla para seleccionar la siguiente línea de código.*

Esta flecha puede arrastrarse a lo largo del código, y tiene la particularidad de que a través de ella podemos saltarnos líneas de código que no queremos que se ejecuten, o en su defecto, podemos ejecutar una misma línea de código dos veces.

Otra opción que se despliega al estar en un breakpoint es la de saltar a una línea de código en específico. Para ello, situamos el cursor del ratón sobre la línea que queremos visualizar, y aparecerán una flecha de color verde al principio de la línea. Al hacer click en ella, el programa se ejecutará hasta esa línea de código en particular.



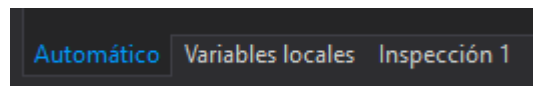
*Fig. 53 – Saltar hasta una línea en particular.*

Este comportamiento también puede ejecutarse a través del teclado, utilizando la combinación **Ctrl + F10**.

### II.2.3 – Visor de Objetos

Los breakpoints son útiles porque nos permiten detener la ejecución del programa, pero realmente su utilidad radica en la posibilidad de utilizar esta interrupción para analizar el estado actual del programa en un momento dado. Es aquí donde surge la utilidad del visor de objetos, que es una herramienta que nos permite visualizar las variables, sus tipos y sus valores en el momento de un breakpoint.

Visual Studio define tres visores de objetos distintos:



*Fig. 54 – Visores de objetos*

- **Automático:** Muestra los valores de las variables circundantes a la línea de código que se está ejecutando, es decir, de las variables cuya modificación fue más reciente en la línea de código. Por ejemplo, en el siguiente breakpoint:

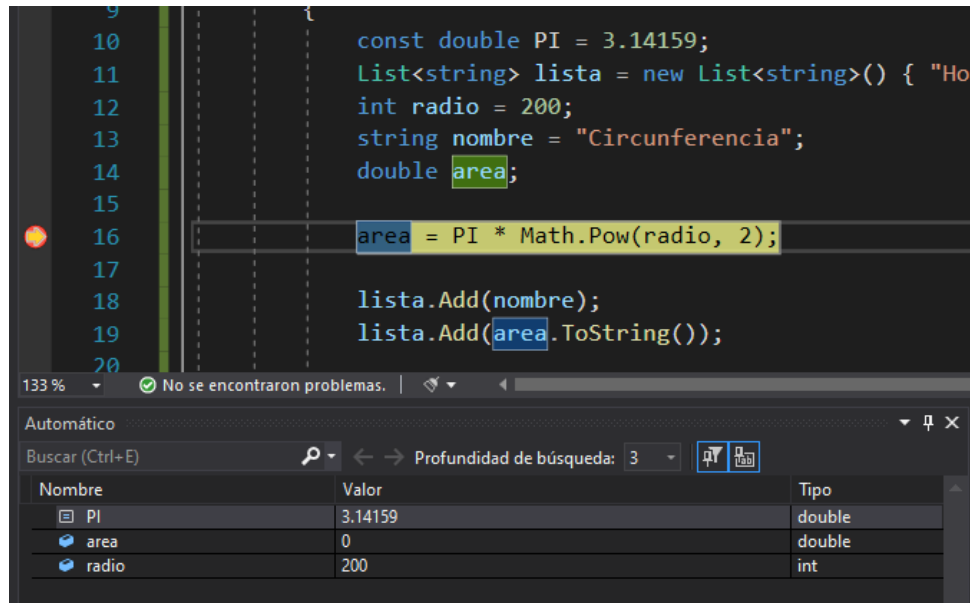


Fig. 55 – Variables tomadas en Automático

En este caso, sólo se están mostrando los valores de PI, área y radio, dado que son las variables que se están manipulando en ese momento en particular. A destacar cómo el valor de área es igual a 0, esto es dado que la interrupción se lleva a cabo *antes* de la ejecución de esta línea de código, en cuyo momento la variable área está definida pero aún no se ha efectuado el cálculo.

- **Variables Locales:** Muestra todas las variables que están en el alcance local del método. Esto es, todas las variables que se recibieron como parámetro, o que se declararon e instanciaron en la ejecución del método. Si la variable no se ha declarado en el momento de la interrupción, esta aparecerá con un valor nulo o cero.

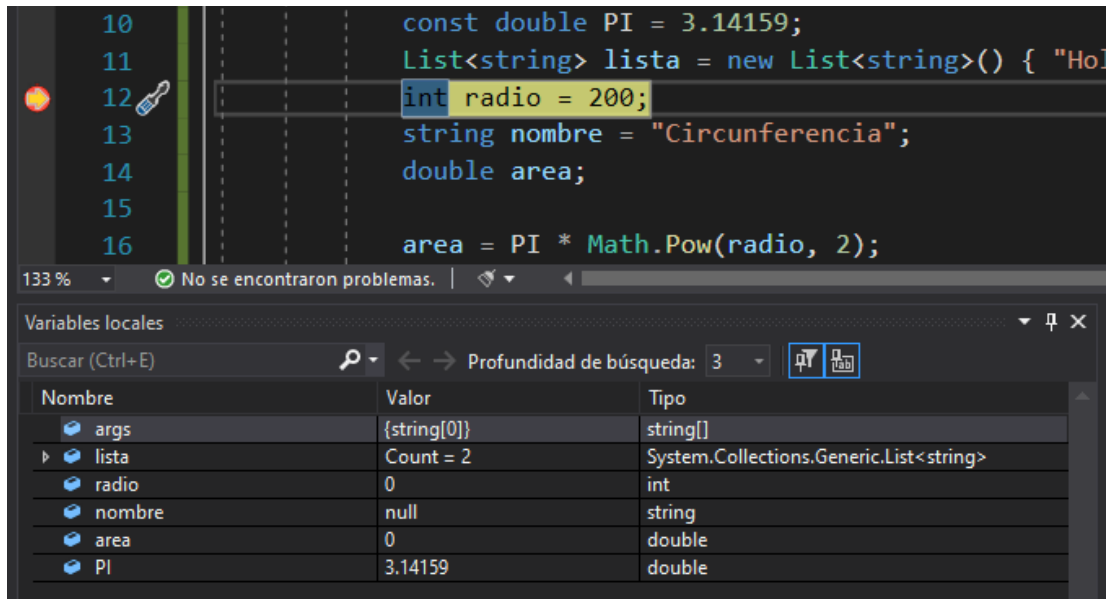
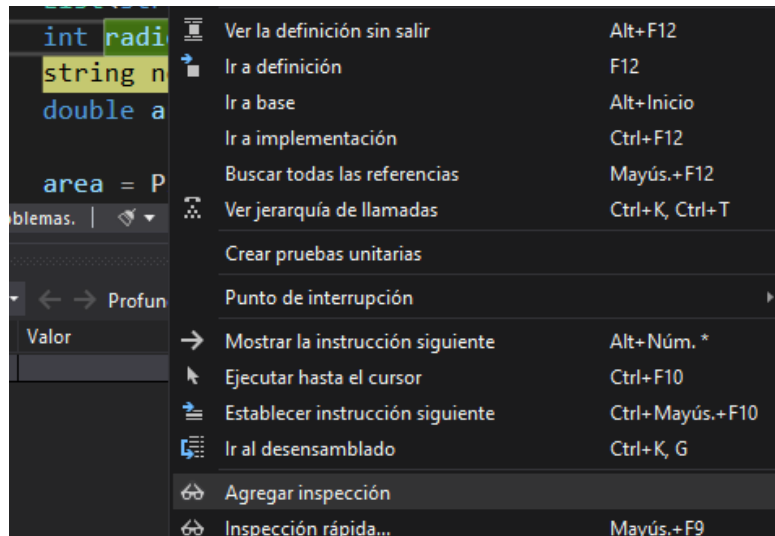


Fig. 56 – Variables locales

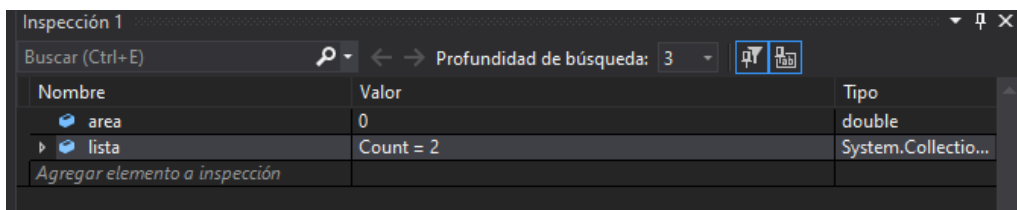
A destacar cómo en el punto de interrupción, las variables `radio`, `nombre` y `área` no se han declarado, pero aún así estas aparecen en el visor de objetos, sólo que con valor 0 o null dependiendo de su tipo.

- **Inspección:** Muestra las variables que yo haya seleccionado previamente para mantener control de las mismas. Para seleccionarlas, debo hacer click derecho en el nombre de la variable – Agregar Inspección.




*Fig. 57 – Agregando inspección*

Al hacer esto, podemos monitorear únicamente las variables que nosotros queramos en todo momento. Por ejemplo, podemos agregar inspección para las variables `área` y `lista`, y tenerlas en el visor de objetos sin mayor problema.



*Fig. 58 – Variables en inspección*

A través del visor de objetos podemos ver, si se trata de un objeto complejo y no primitivo, los detalles de todos sus atributos en cualquier momento al hacer click en el ícono  situado a la izquierda de la variable.

Por ejemplo, si lo hacemos con la variable lista, tendremos la siguiente vista:

lista	Count = 2	System.Collectio
[0]	"Hola"	string
[1]	"Depuración"	string
Raw View		

*Fig. 59 – Vista del List<string>*

En este caso, al tratarse de un List, tendremos vista de todos los objetos que conforman la misma. Si hacemos click sobre Raw View, tendremos una vista de las propiedades, atributos y métodos de la misma:

Raw View		
Capacity	4	int
Count	2	int
Static members		
Non-Public members		
System.Collections.G...	false	bool
System.Collections.I...	false	bool
System.Collections.I...	Count = 2	object (System.C...
System.Collections.IL...	false	bool
System.Collections.IL...	false	bool
_items	{string[4]}	string[]
_size	2	int
_version	2	int

*Fig. 60 – Raw View del List<string>*

Ahora, asumamos la siguiente clase e instancia de un Estudiante:

```
class Estudiante
{
    public string nombre { set; get; }
    public int edad { set; get; }
    public DateTime fechaNacimiento { set; get; }
    public string numeroTelefono { set; get; }
}

Estudiante es = new Estudiante()
{
    nombre = "Gabriel",
    edad = 24,
    fechaNacimiento = new DateTime(1994, 7, 1),
    numeroTelefono = "0414-1112233"
};
```

Fig. 61 – Clase e instancia de un Estudiante

En el visor de objetos, es aparecería así:

es	{EjemploDebug.Estudiante}	EjemploDebug.Estudiante
edad	24	int
fechaNacimiento	{01/07/1994 12:00:00 }	System.DateTime
nombre	"Gabriel"	string
numeroTelefono	"0414-1112233"	string

Fig. 62 – Muestra de objeto en el visor de objetos.

Si quisiéramos ver el estado actual de una variable sin necesidad de consultar el visor de objetos, tendríamos dos opciones:

- 1) Situar el cursor del ratón sobre la variable a visualizar, lo que ejecutará un pequeño visor de objetos con el que podremos interactuar.

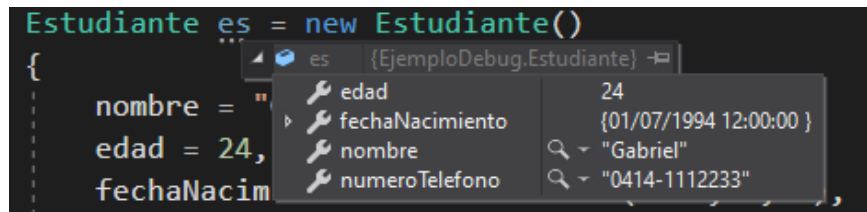


Fig. 63 – Visor de objetos sobre el editor de texto

- 2) Hacer click derecho – Inspección rápida (o seleccionar la variable y presionar **Shift + F9**)

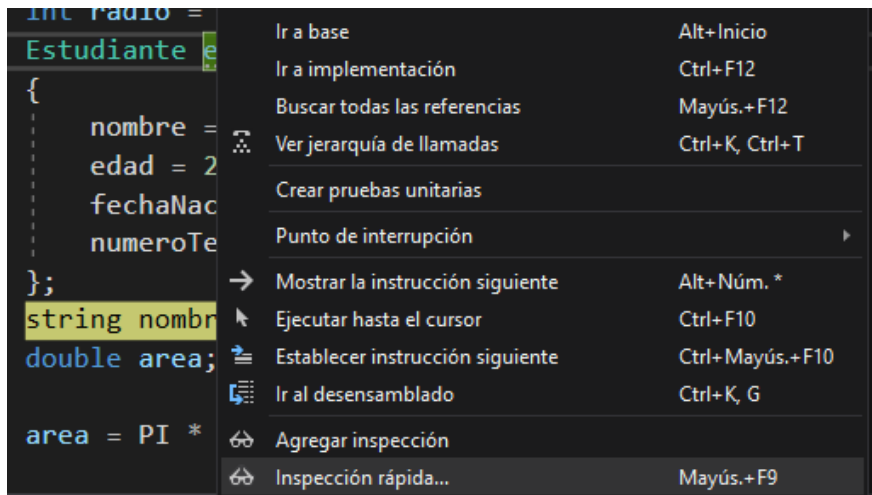
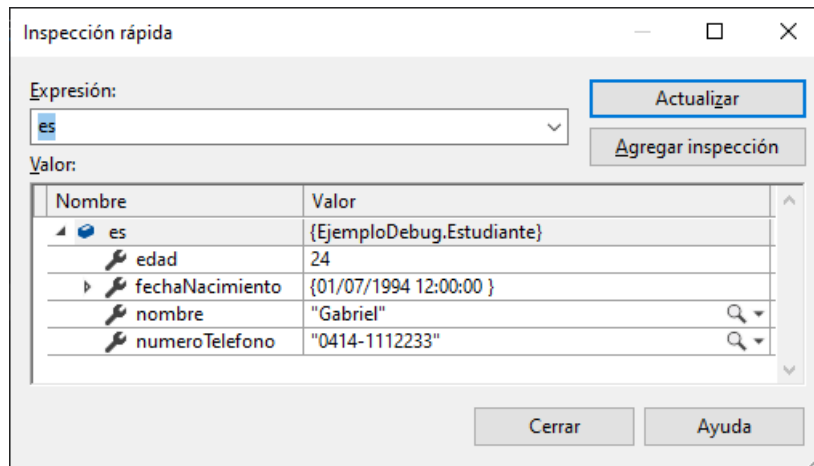


Fig. 64 – Inspección rápida



Al hacer esto, se abrirá una ventana que mostrará una inspección rápida del objeto.



*Fig. 65 – Ventana de Inspección rápida*

En cualquier área del visor de objetos, también podremos modificar los valores de las variables, al hacer doble click en la celda correspondiente al valor de una variable en particular. Este cambio hará efecto inmediatamente, por lo que debemos ser cuidadosos ya que esto también podría generar efectos inesperados en el programa.

### III – Referencias Bibliográficas

Farrell, J. (2018). *Microsoft Visual C# 2017: An Introduction to Object-Oriented Programming*. Boston: Cengage Learning.

Griffiths, I. (2019). *Programming C# 8.0*. Sebastopol: O'Reilly Media, Inc.

Microsoft. (28 de 01 de 2021). *C# documentation*. Obtenido de Microsoft Docs: <https://docs.microsoft.com/en-us/dotnet/csharp/>

Siahaan, V. (2020). *VISUAL C# .NET FOR STUDENTS: A Project-Based Approach to Develop Desktop Applications*. Balige: BALIGE PUBLISHING.

Siahaan, V., & Sianipar, R. (2020). *VISUAL C# .NET: A Step By Step, Project-Based Guide to Develop Desktop Applications*. Balige: BALIGE PUBLISHING.

Villalobos, J., & Casallas, R. (s.f.). *Fundamentos de Programación. Aprendizaje activo basado en casos*. Bogotá: Universidad de los Andes - Facultad de Ingeniería.