Introducción a C#

Manual del estudiante

Miguel Muñoz Serafín

Agosto, 2017

TI Capacitación

Introducción a C#

Manual de estudiante

Primera edición

Agosto de 2017

Soporte técnico:
soporte@mail.ticapacitacion.com
https://ticapacitacion.com/promociones/introcs.html

Este manual fue creado por TI Capacitación para uso personal de:

Luis Esteban García Rosas
estebangaro@outlook.com

Contenido

Introducción a C#

Introducción

Acerca del curso

Este curso proporciona a los participantes, los conocimientos y habilidades requeridas para crear aplicaciones utilizando el lenguaje de programación C#.

El curso inicia con una introducción a la estructura básica de una aplicación C# y la sintaxis del lenguaje para posteriormente describir las distintas características proporcionadas por el .NET Framework, así como las técnicas y tecnologías empleadas para desarrollar aplicaciones de escritorio y empresariales.

Al final del curso, los participantes tendrán los conocimientos básicos del lenguaje de programación C# para desarrollar aplicaciones .NET.

Audiencia

Este curso está dirigido a todas las personas que se encuentren interesadas en aprender el lenguaje de programación C# y que cuenten con conocimientos básicos en algún lenguaje de programación como C, C++, JavaScript, Objective-C o Java entre otros.

Objetivos

Al finalizar este entrenamiento los participantes serán capaces de:

- Describir la sintaxis fundamental y características de C#.
- Crear e invocar métodos.
- Administrar Excepciones.
- Describir los requerimientos para monitoreo de aplicaciones de escala empresarial.
- Implementar la estructura básica y los elementos esenciales de una aplicación de escritorio tradicional.
- Crear Clases.
- Definir e implementar Interfaces.
- Crear y utilizar colecciones genéricas.
- Utilizar herencia para crear una jerarquía de Clases.
- Extender una Clase del .NET Framework.
- Crear Clases y métodos genéricos.

Requerimientos

Para poder practicar los conceptos y realizar los ejercicios del curso, se recomienda trabajar con un equipo de desarrollo con Windows 10 y Visual Studio 2015 o posteriores. Puede utilizarse la versión gratuita **Visual Studio Community** que puede descargarse desde el siguiente enlace:

https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

Los videos de demostración de este curso fueron realizados con Visual Studio 2015.

Contenido del curso

El contenido de este entrenamiento está dividido en 5 módulos.

Módulo 1: Introducción a C#

En este módulo, se examina la sintaxis y características principales del lenguaje de programación C#, proporcionando también, una introducción al depurador de Visual Studio.

Al finalizar este módulo, los participantes podrán:

- Describir la arquitectura de las aplicaciones .NET y utilizar las características que Visual Studio y C# proporcionan para dar soporte al desarrollo de aplicaciones con el .NET Framework.
- Utilizar los tipos de datos básicos, operadores y expresiones proporcionadas por C#.
- Utilizar las sentencias principales del lenguaje de programación C#.

Los temas que forman parte de este módulo son:

- Lección 1: Desarrollando aplicaciones con C#
- Lección 2: Tipos de datos, Operadores y Expresiones
- Lección 3: Sentencias principales del lenguaje de programación C#

Módulo 2: Creación de Métodos, Manejo de Excepciones y Monitoreo de Aplicaciones

En este módulo se explica la forma de crear e invocar métodos, así como la manera de capturar y manejar Excepciones. Se describe también los requerimientos para monitorear aplicaciones de alta escalabilidad.

Al finalizar este módulo, los participantes podrán:

- Crear e invocar métodos, pasar parámetros a los métodos y devolver valores desde los métodos.
- Crear métodos sobrecargados y utilizar parámetros opcionales y parámetros de salida (output parameters).

- Capturar y manejar Excepciones, así como escribir información al log de eventos de Windows.
- Explicar los requerimientos para implementar el registro (Logging), seguimiento (Tracing) y análisis de rendimiento (Profiling) cuando se construyen aplicaciones de gran escala.

Los temas que se cubren en este módulo son:

- Lección 1: Creando e invocando Métodos
- Lección 2: Creando Métodos sobrecargados y utilizando parámetros opcionales y de salida (output)
- Lección 3: Manejo de Excepciones
- Lección 4: Monitoreo de aplicaciones

Módulo 3: Desarrollando el código para una aplicación gráfica

En este módulo, se describe la forma de implementar la estructura básica y los elementos esenciales de una aplicación gráfica típica, incluyendo el uso de estructuras, enumeraciones, colecciones y eventos.

Al finalizar este módulo, los participantes podrán:

- Definir y utilizar estructuras y enumeraciones.
- Crear y utilizar colecciones simples para almacenar datos en memoria.
- Crear, suscribir y lanzar eventos.

Los temas que se cubren en este módulo son:

- Lección 1: Implementando Estructuras y Enumeraciones
- Lección 2: Organizando datos dentro de colecciones
- Lección 3: Manejando Eventos

Módulo 4: Creando Clases e implementando colecciones de Tipos Seguros (Type-safe collections)

En este módulo se explica cómo crear Clases, definir e implementar Interfaces, así como la forma de crear y utilizar colecciones genéricas. Se describe también las diferencias entre Tipos Valor (Value Type) y Tipos Referencia (Reference Type) en C#.

Al finalizar este módulo, los participantes podrán:

- Crear y utilizar Clases personalizadas.
- Definir e implementar Interfaces personalizadas.
- Utilizar Tipos Genéricos para implementar colecciones de tipos seguros (Type-safe Collections).

Los temas que se cubren en este módulo son:

- Lección 1: Creando Clases
- Lección 2: Definiendo e implementando Interfaces
- Lección 3. Implementando colecciones de tipos seguros (Type-safe Collections)

Módulo 5: Creando una jerarquía de Clases utilizando Herencia

En este módulo se explica cómo utilizar herencia para crear una jerarquía de Clases y la manera de extender una Clase del .NET Framework. Este módulo también describe la forma de crear Clases Genéricas y la forma de definir Métodos de Extensión.

Al finalizar este módulo los participantes podrán:

- Definir Clases Abstractas.
- Heredar desde Clases Base para crear una jerarquía de Clases.
- Heredar desde Clases del .NET Framework.
- Utilizar Métodos de Extensión para agregar funcionalidad personalizada a las clases heredadas.
- Crear Clases y Métodos Genéricos.

Los temas que se cubren en este módulo son:

- Lección 1: Creando Jerarquías de Clases
- Lección 2: Extendiendo Clases del .NET Framework

Introducción a C#

Módulo 1: Introducción a C#

Acerca del módulo

El Microsoft .NET Framework proporciona una plataforma de desarrollo que podemos utilizar para construir, desplegar y administrar aplicaciones y servicios. En este módulo, conoceremos las principales características proporcionadas por el .NET Framework y Microsoft Visual Studio.

Examinaremos algunas de las sentencias clave de Visual C# que nos permitirán empezar a desarrollar aplicaciones para el .NET Framework.

Objetivos

Al finalizar este módulo, los participantes contarán con las habilidades y conocimientos para:

- Describir la arquitectura de las aplicaciones .NET
- Utilizar las características que Visual Studio y C# proporcionan para dar soporte al desarrollo de aplicaciones para el .NET Framework.
- Utilizar los tipos de datos básicos, operadores y expresiones proporcionadas por C#
- Utilizar las sentencias principales del lenguaje de programación C#.

Los temas que se cubren en este módulo son:

- Lección 1: Desarrollando aplicaciones con C#.
- Lección 2: Tipos de datos, Operadores y Expresiones.
- Lección 3: Sentencias principales del lenguaje de programación C#.

Lección 1:
Desarrollando aplicaciones con C#

El .NET Framework y Visual Studio proporcionan diversas características que podemos utilizar cuando desarrollamos nuestras aplicaciones.

En esta lección, aprenderemos acerca de las características que Visual Studio y el .NET Framework proporcionan y que nos permiten crear nuestras propias aplicaciones.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Describir la plataforma Microsoft .NET.
- Describir el propósito del .NET Framework.
- Describir la arquitectura de las aplicaciones .NET.
- Describir las características clave de Visual Studio.
- Describir las plantillas de proyectos proporcionadas en Visual Studio.
- Crear una aplicación .NET Framework.
- Describir XAML.

¿Qué es Microsoft .NET?

Microsoft .NET es una propuesta de Microsoft que proporciona una plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma de hardware y que permite un rápido desarrollo de aplicaciones. Con esta propuesta, se facilita la interconexión de distintas plataformas de hardware, software, información y usuarios. Basado en esta plataforma, se ha desarrollado una estrategia horizontal que integra productos que van desde sistemas operativos, herramientas de desarrollo hasta las aplicaciones para usuario final.

La plataforma .NET no es un producto empaquetado que se pueda comprar como tal, sino que es una plataforma que engloba distintas aplicaciones, servicios y conceptos y que en conjunto permiten el desarrollo y la ejecución de aplicaciones.

Microsoft .NET puede considerarse como una respuesta de Microsoft al creciente mercado de los negocios en entornos Web. La propuesta es ofrecer una manera rápida y económica, a la vez que segura y robusta, de desarrollar aplicaciones permitiendo una integración más rápida y ágil entre empresas y un acceso más simple y universal a todo tipo de información desde cualquier tipo de dispositivo.

La plataforma .NET proporciona un modelo de programación consistente e independiente del lenguaje en todas las capas de una aplicación. Permite la interoperabilidad y una fácil migración de las tecnologías existentes.

La plataforma .NET proporciona nuevas formas de construir aplicaciones a partir de colecciones de servicios Web. Soporta completamente la infraestructura actual de Internet incluyendo HTTP, XML y SOAP entre otros estándares.

La plataforma .NET está compuesta por un conjunto de tecnologías diseñadas para transformar a Internet en una plataforma de cómputo distribuido. Proporciona Interfaces de programación y herramientas para diseñar, crear, ejecutar y distribuir soluciones para la plataforma .NET, por ejemplo, Visual Studio.

Dentro de la plataforma .NET se encuentran dispositivos ejecutando sistemas operativos que se integran e interactúan con otros elementos .NET. Se incluyen dispositivos móviles, Navegadores Web, Xbox, PCs, etc.

Los servidores también forman parte de la plataforma Microsoft .NET y proporcionan la infraestructura para administrar, construir y distribuir soluciones para la plataforma .NET, por ejemplo, Windows Server, SQL Server o Exchange Server.

Los Servicios Web, que son un conjunto de servicios predefinidos que realizan tareas específicas y que permiten a los desarrolladores crear sus propios servicios, también forman parte de la plataforma .NET

Las experiencias de usuario también forman parte de la plataforma Microsoft .NET y se componen de Software integrado con servicios XML Web que presentan al usuario la información que necesita y en la forma en que la necesita.

Componentes Principales

Los elementos principales de la plataforma .NET incluyen:

- **El .NET Framework.** El .NET Framework proporciona un entorno de ejecución de aplicaciones denominado **Common Language Runtime** o **CLR**. Este entorno es un componente de software cuya función es la de ejecutar las aplicaciones .NET e interactuar con el sistema operativo, ofreciendo sus servicios y recursos a estas aplicaciones .NET. Este entorno es común a todos los lenguajes .NET. Con esto podemos decir que la plataforma .NET no sólo nos brinda todas las herramientas y servicios que se necesitan para desarrollar modernas aplicaciones empresariales y de misión crítica, sino que también nos provee de mecanismos robustos, seguros y eficientes para asegurar que la ejecución de las mismas sea óptima.

- **Common Languaje Specification**. CLS define los estándares comunes a los que los lenguajes y desarrolladores deben adherirse para que sus componentes y aplicaciones puedan ser utilizados por otros lenguajes compatibles con .NET. El CLS permite que desarrolladores de Visual Basic puedan trabajar en equipo con otros desarrolladores de Visual C# sin que esto genere problemas de interoperabilidad entre el código generado por ambos lenguajes. Un desarrollador Visual Basic, por ejemplo, podrá consumir clases desarrolladas en Visual C# y viceversa.

- **Servidores empresariales .NET**. Los servidores empresariales .NET proporcionan escalabilidad, confiabilidad, administración e integración dentro de la empresa o incluso entre distintas empresas. Dentro de los servidores .NET podemos encontrar a Windows Server, SQL Server, Bistalk Server, Exchange Server, Sharepoint Server, etc.

- **Bloque de Servicios**. Los bloques de servicios están formados por Servicios Web que ofrecen diversas funcionalidades. Como ejemplo de estos bloques de servicios podemos mencionar al servicio *Microsoft Account* (anteriormente conocido como Windows Live) o al servicio OneDrive (anteriormente conocido como *SkyDrive)* entre otros.

- **Visual Studio**. Visual Studio .NET proporciona un ambiente para desarrollar aplicaciones para el .NET Framework. Visual Studio simplifica el proceso entero de desarrollo de software, desde el diseño hasta la implementación. Visual Studio tiene soporte para múltiples monitores, desarrollo de aplicaciones para Sharepoint, permite desarrollar aplicaciones para distintas versiones del .NET Framework, Soporte Intellisense y mucho más.

Como parte de la plataforma .NET, también contamos con un conjunto de lenguajes de programación de alto nivel, junto con sus compiladores y linkers, que permiten el desarrollo de aplicaciones para la plataforma .NET.

¿Qué es el .NET Framework?

El .NET Framework es la plataforma de desarrollo de código administrado de Microsoft. Está formado por una serie de herramientas y librerías con las que se pueden crear todo tipo de aplicaciones, desde las tradicionales aplicaciones de escritorio hasta aplicaciones para XBOX pasando por desarrollo Web, desarrollo para el Windows Store y Windows Phone así como aplicaciones de servidor con WCF.

El .NET Framework está compuesto por un conjunto de tecnologías que forman una parte importante de la plataforma .NET. Constituye una infraestructura de programación para construir, distribuir y ejecutar aplicaciones y servicios para la plataforma .NET.

El .NET Framework soporta completamente las características de la programación orientada a objetos. Soporta el uso de Herencia, Polimorfismo, Clases, propiedades, métodos, eventos, constructores y otras estructuras de la programación orientada a objetos.

Proporciona un ambiente que minimiza los conflictos de versiones de los DLL´s (DLL Hell) a los que se enfrentan los programadores que utilizan los componentes COM y simplifica la distribución e instalación de las aplicaciones.

Proporciona un ambiente de portabilidad basado en estándares certificados que permiten que las aplicaciones puedan ser hospedadas por cualquier sistema operativo. Actualmente C# y la mayor parte del motor de ejecución de .NET conocido como CLR y que es una implementación del Common Language Infrastructure (CLI) han sido estandarizados por ECMA. (European Computer Manufacturers Association).

El .NET Framework proporciona un ambiente administrado (Managed) en el cual el código es verificado para realizar una ejecución segura. Ofrece un entorno de ejecución de código que fomenta la ejecución segura del mismo, incluso del código creado por terceras personas desconocidas o que no son de plena confianza.

Componentes Principales del .NET Framework

El .NET Framework proporciona 3 elementos principales, el **Common Language Runtime** o motor en tiempo de ejecución común para todos los lenguajes .NET, **El .Net Framework Class Library** o biblioteca de clases base del .NET Framework y una **Colección de Frameworks** de desarrollo.

El **Common Language Runtime** es el corazón del .NET Framework. El CLR como comúnmente se le conoce, es la implementación de Microsoft del CLI. Es el agente encargado de administrar la ejecución del código y simplifica el proceso de desarrollo, proporcionando un ambiente de ejecución robusto y altamente seguro con servicios centrales como la compilación en tiempo de ejecución, Administración de memoria, Seguridad, Administración de los hilos de ejecución (Threads) además

de encargarse de aplicar una seguridad estricta a los tipos de datos y la interoperabilidad con código no administrado.

El concepto de administración de código es un principio básico del motor en tiempo de ejecución. El código que corre sobre el **CLR** es conocido como código administrado (Managed Code). Existen aplicaciones tales como componentes COM y aplicaciones basadas en las APIs de Windows cuyo código no requiere del ambiente administrado que ofrece el CLR. Este tipo de código recibe el nombre de código no administrado (Unmanaged Code).

La **Biblioteca de clases base del .NET Framework**, proporciona una colección completa orientada a objetos de tipos reutilizables que contiene Clases y estructuras de datos que se pueden emplear para desarrollar prácticamente todo tipo de aplicaciones. Las clases proporcionan la base de la funcionalidad común y elementos que ayudan a simplificar el desarrollo de aplicaciones, eliminando la necesidad de reinventar la lógica constantemente. Por ejemplo, la clase **System.IO.File** contiene funcionalidad que nos permite manipular el sistema de archivos de Windows. Además de utilizar las clases de la biblioteca de clases base, podemos extender estas clases, creando nuestras propias bibliotecas de clases.

El .NET Framework proporciona varios Frameworks de desarrollo que podemos utilizar para construir los tipos de aplicaciones comunes, incluyendo:

- Aplicaciones de escritorio cliente mediante el uso de Windows Presentation Foundation (WPF).
- Aplicaciones de escritorio de Windows Store utilizando XAML.
- Aplicaciones Web del lado del servidor, mediante ASP.NET Web Forms o ASP.NET MVC.
- Aplicaciones Web orientadas a servicios, mediante el uso de Windows Communication Foundation (WCF) o ASP.NET MVC Web API.
- Aplicaciones de ejecución en segundo plano mediante el uso de servicios Windows.

Cada Framework proporciona los componentes y la infraestructura necesaria para desarrollar aplicaciones.

El .NET Framework, traducido como “Marco de Trabajo”, es el componente fundamental de la plataforma Microsoft .NET, necesario tanto para poder desarrollar aplicaciones como para poder ejecutarlas luego en entornos de prueba o producción.

El CLR se comunica con el sistema operativo para proporcionar recursos a las aplicaciones administradas.

La biblioteca de clases base del .NET Framework, proporciona una colección completa orientada a objetos de tipos reutilizables que contiene Clases y Estructuras de datos que se pueden emplear para desarrollar prácticamente todo tipo de aplicaciones administradas.

Un tipo especial de aplicaciones administradas son las bibliotecas de clases personalizadas.

Las aplicaciones administradas pueden acceder a la biblioteca de clases base y a las bibliotecas de clases personalizadas.

Las aplicaciones administradas no se comunican directamente con el sistema operativo.

Las aplicaciones no administradas se comunican con el sistema operativo y pueden convivir con las aplicaciones administradas.

Los servicios de **Internet Information Server (IIS)** dan soporte a las aplicaciones ASP.NET. ASP.NET es el soporte para las aplicaciones Web administradas.

El motor ASP.NET se comunica con Internet Information Server y no con el sistema operativo.

Para obtener más información sobre el .**NET Framework**, se recomienda visitar el siguiente enlace:

Overview of the .NET Framework
https://msdn.microsoft.com/en-us/library/zw4w595w.aspx

El .NET Framework y el estándar CLI

El **CLI** (Common Language Infrastructure) define el ambiente virtual de ejecución de código que es independiente de cualquier plataforma. Como mencionamos anteriormente, ha sido estandarizado por ECMA. No es específica de un sistema operativo en particular.

El **CLR** del .NET Framework es la implementación del Common Language Infrastructure. El .NET Framework contiene más características que las especificadas en la arquitectura del CLI.

Una de las preocupaciones que tiene un desarrollador que decide invertir su tiempo en aprender C# y .NET es saber si esos conocimientos le permitirán desarrollar aplicaciones para otras plataformas. La pregunta típica es ¿El .NET Framework es un producto Microsoft diseñado únicamente para el sistema operativo Windows o es una plataforma que permite transportar las aplicaciones .NET a otros sistemas operativos?

Debido a que el CLI es independiente de cualquier plataforma y no es específica de un sistema operativo en particular, una aplicación .NET podría ejecutarse transparentemente en un ambiente Windows o Linux.

La parte central del CLI es la definición de un lenguaje intermedio común – Common Intermediate Language (CIL) – que debe ser generado por los compiladores compatibles con el CLI y un sistema de tipos de datos que define los distintos tipos de datos soportados por cualquier lenguaje compatible.

El CLI incluye además los estándares para el lenguaje C# desarrollado por Microsoft. Otros proveedores que han adoptado el estándar CIL, han desarrollado compiladores .NET para lenguajes como Python, Pascal, Fortran, Cobol e Eiffel entre otros.

Caracteristicas principales de Visual Studio

Visual Studio proporciona un ambiente de desarrollo que permite diseñar, implementar, compilar, probar y desplegar rápidamente varios tipos de aplicaciones y componentes utilizando una amplia gama de lenguajes de programación.

Algunas de las principales características de Visual Studio son:

- **Un Entorno de desarrollo integrado (IDE) intuitivo**. El IDE de Visual Studio ofrece todas las características y herramientas que son necesarias para diseñar, implementar, desarrollar, probar e implementar aplicaciones y componentes.
- **Desarrollo rápido de aplicaciones**. Visual Studio ofrece vistas de diseño para componentes gráficos que permiten crear fácilmente interfaces de usuario complejas. Alternativamente, podemos utilizar las vistas del editor de código, que proporcionan un mayor control. Visual Studio también proporciona asistentes que ayudan a acelerar el desarrollo de componentes particulares.
- **Acceso a datos y servidores**. Visual Studio proporciona el Explorador de servidores, que nos permite iniciar sesión en los servidores y explorar sus bases de datos y servicios del sistema. También proporciona una manera familiar para crear, consultar y modificar bases de datos que la aplicación utiliza a través del diseñador de tablas.
- **Internet Information Server (IIS) Express**. Visual Studio proporciona una versión ligera de IIS como servidor Web predeterminado para la depuración de nuestras aplicaciones Web.
- **Funciones de Depuración**. Visual Studio proporciona un depurador que permite ejecutar paso a paso a través de código local o remoto, hacer una pausa en los puntos de interrupción y seguir rutas de ejecución.
- **Manejo de errores**. Visual Studio proporciona la ventana **Lista de errores**, que muestra los errores, advertencias o mensajes que se producen al editar y crear nuestro código.
- **Ayuda y Documentación**. Visual Studio ofrece ayuda y orientación a través del **Microsoft IntelliSense**, fragmentos de código y el sistema de ayuda integrada que contiene documentación y ejemplos.

Para obtener más información acerca de las novedades en Visual Studio, se recomienda visitar el siguiente enlace:

Novedades de Visual Studio 2015
https://msdn.microsoft.com/es-mx/library/bb386063.aspx

Plantillas de Visual Studio

Visual Studio nos proporciona un entorno de Desarrollo que permite crear aplicaciones Windows, Web, Servicios, Bibliotecas, o aplicaciones empresariales para la nube. Para ayudarnos a iniciar, Visual Studio proporciona una serie de plantillas de aplicación que proporcionan una estructura para los diferentes tipos de aplicaciones. Las plantillas:

- Proporcionan el código inicial para construir y crear rápidamente aplicaciones funcionales.
- Incluyen soporte a componentes y controles dependiendo del tipo de proyecto seleccionado.
- Configuran el IDE de Visual Studio dependiendo del tipo de aplicación que se desea desarrollar.
- Agregan referencias a los ensamblados que el tipo de aplicación requiera.

La siguiente tabla describe algunas de las plantillas comunes de aplicaciones que podríamos utilizar cuando desarrollemos aplicaciones .NET Framework utilizando Visual Studio.

Plantilla	Descripción
Console Application	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una aplicación que se ejecuta en una interfaz de línea de comandos. Este tipo de aplicación es considerada ligera debido a que no tiene una interfaz de usuario gráfica.
Windows Forms Application	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una aplicación gráfica Windows Form.
WPF Application	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una aplicación Windows rica en interfaz de usuario. Una aplicación WPF nos permite crear la siguiente generación de aplicaciones Windows con mucho más control sobre el diseño de la interfaz de usuario.
Blank App (Universal Windows)	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una aplicación de la Plataforma Universal de Windows (Universal Windows Platform).
Blank App (Universal Windows 8.1)	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una aplicación Universal para Windows y Windows Phone que utiliza el Windows Runtime.
Class Library	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar una biblioteca de clases. Esta plantilla nos permite generar un Assembly .dll que podemos utilizar para almacenar funcionalidad que podríamos querer invocar desde otra aplicación.
ASP.NET Web Application (.NET Framework)	Proporciona la configuración del ambiente, herramientas, referencias y código inicial para desarrollar aplicaciones ASP.NET. Podemos crear aplicaciones Web Forms, MVC, Web API y otro tipo de aplicaciones ASP.NET.

Introducción a XAML

Extensible Application Markup Language (**XAML)**, es un lenguaje basado en XML que podemos utilizar para definir la interfaz de usuario de aplicaciones .NET. XAML también puede ser utilizado para desarrollar aplicaciones para la **Plataforma Universal de Windows (UWP)** o para desarrollar aplicaciones para iOS y Android con Xamarin.

XAML utiliza elementos y atributos para definir controles y sus propiedades en sintaxis XML. Cuando diseñamos una interfaz de usuario, podemos utilizar la caja de herramientas y el panel de propiedades de Visual Studio para crear visualmente la interfaz de usuario. Podemos utilizar el panel XAML para crear la interfaz de usuario de forma declarativa. También podemos utilizar **Blend** para Visual Studio o cualquier herramienta de terceros. Algunos desarrolladores pueden sentirse más cómodos al escribir directamente código **XAML** sobre el *panel XAML* en lugar de arrastrar los controles desde el cuadro de herramientas hacia la ventana.

El siguiente ejemplo muestra la declaración XAML de una etiqueta, una caja de texto y un botón.

```
<TextBlock Text="Name:" HorizontalAlignment="Left" Margin="72,43,0,0"
    VerticalAlignment="Top" />
<TextBox HorizontalAlignment="Left" Height="23" Margin="141,43,0,0" Text=""
      VerticalAlignment="Top" Width="120" />
<Button Content="Click Me!" HorizontalAlignment="Left" Margin="119,84,0,0"
     VerticalAlignment="Top" Width="75" />
```

El definir la interfaz de usuario en XAML, nos permite que esta sea más portable y separa la interfaz de usuario de la lógica de la aplicación. Por ejemplo, es posible migrar una gran parte del código XAML de una aplicación WPF y reutilizarla en una aplicación UWP o Xamarin.

Podemos utilizar la sintaxis XAML para generar interfaces de usuario simples o construir interfaces de usuario mucho más complejas. La sintaxis de XAML proporciona la funcionalidad para enlazar datos a controles, utilizar gradientes, texturas, plantillas para dar formato a los datos y enlazar a eventos de los controles.

También es posible utilizar distintos contenedores para posicionar los controles de nuestra interfaz de usuario de forma apropiada y dar una apariencia uniforme ajustándose al tamaño de la pantalla.

Lección 2:
Tipos de datos, Operadores y Expresiones

La mayoría de las aplicaciones utilizan datos, estos datos podrían ser proporcionados por el usuario a través de una interfaz de usuario, desde una base de datos, desde un servicio de red, o desde alguna otra fuente. Para almacenar y utilizar datos en nuestras aplicaciones, debemos familiarizarnos con la forma de definir y utilizar variables, así como en la forma de crear y utilizar expresiones con la variedad de operadores que Visual C# proporciona.

En esta lección, conoceremos la forma de utilizar algunos de los elementos fundamentales de la programación en Visual C#, tales como variables, miembros de los tipos de datos, conversiones y manipulación de cadenas.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Describir los tipos de datos proporcionados por Visual C#.
- Crear y utilizar expresiones.
- Declarar y asignar variables.
- Acceder a los miembros de la instancia de un tipo de dato.
- Convertir datos de un tipo a otro.
- Concatenar y validar cadenas.

¿Qué son los tipos de Datos?

Una variable contiene datos de un tipo específico. Cuando declaramos una variable para almacenar los datos en una aplicación, debemos elegir un tipo de dato adecuado para los datos. Visual C# es un lenguaje de tipos seguros "**Type-Safe**", esto significa que el compilador garantiza que los valores almacenados en las variables siempre son del tipo apropiado.

La siguiente tabla muestra los tipos de datos más comunes que son utilizados en Visual C#.

Tipo	Descripción	Tamaño en bytes	Rango
int	Números enteros.	4	–2,147,483,648 a 2,147,483,647
long	Números enteros.	8	–9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	Números de punto flotante.	4	+/–3.4 × 10^38
double	Números de punto flotante de doble precisión (más precisos).	8	+/–1.7 × 10^308
decimal	Valores de moneda.	16	-7.9 x 10^28 a 7.9 x 10^28
char	Un simple carácter Unicode.	2	N/A
bool	Valor booleano.	1	Falso o Verdadero.
DateTime	Momentos en el tiempo	8	0:00:00 del 01/01/2001 a 23:59:59 del 12/31/9999
string	Secuencia de caracteres.	2 por carácter.	N/A

Para mayor información acerca de los tipos de datos, se recomienda visitar el siguiente enlace:

Reference Tables for Types (C# Reference)
https://msdn.microsoft.com/en-us/library/1dhd7f2x.aspx

Expresiones y Operadores en Visual C#

Las expresiones son el componente central de prácticamente cualquier aplicación Visual C# debido a que las expresiones son construcciones fundamentales que utilizamos para evaluar y manipular datos.

Las expresiones son colecciones de operandos y operadores que podemos definir de la siguiente manera:

- Los operandos son valores por ejemplo números y cadenas. Los operandos pueden ser valores constantes (literales), variables, propiedades o valores devueltos por las llamadas a métodos.
- Los operadores definen operaciones a realizar sobre los operandos, por ejemplo, la suma o la multiplicación. Los operadores existen para todas las operaciones matemáticas básicas, así como para operaciones más avanzadas tales como comparaciones lógicas o la manipulación de bits de datos que constituyen un valor.

Todas las expresiones son evaluadas a un simple valor cuando la aplicación es ejecutada. El tipo de valor que una expresión genera depende de los tipos de operandos y operadores que utilicemos. No existe un límite en la longitud de las expresiones en las aplicaciones Visual C#, aunque en la práctica estamos limitados por la memoria de la computadora y nuestra paciencia al escribir. La recomendación es utilizar expresiones cortas y ensamblar el resultado de las expresiones individuales. Esto nos facilita ver lo que el código está realizando y facilita también la depuración del código.

Operadores en Visual C#

Los operadores se combinan con los operandos para formar expresiones. Visual C# proporciona una amplia gama de operadores que podemos utilizar para realizar las operaciones matemáticas y lógicas fundamentales más comunes. Los operadores caen dentro de una de las siguientes tres categorías:

- **Unario**. Este tipo de operador, opera sobre un solo operando. Por ejemplo, podemos utilizar el operador **–** como un operador unario. Para hacer esto, lo colocamos inmediatamente antes de un operando numérico y el operador convierte el valor del operando a su valor actual multiplicado por **-1**.
- **Binario**. Este tipo de operador opera sobre 2 valores. Este es el tipo más común de operador, por ejemplo, *****, el cual multiplica el valor de dos operandos.
- **Ternario**. Existe un solo operador ternario en Visual C#. Este es el operador **? :** que es utilizado en expresiones condicionales.

La siguiente tabla muestra los operadores más comunes que podemos utilizar en Visual C# agrupados por tipo.

Tipo	Operadores
Aritméticos	+, -, *, /, %
Incremento, decremento	++, --
Comparación	==, !=, <, >, <=, >=, is
Concatenación de cadenas	+
Operaciones lógicas de bits	&, |, ^, !, ~, &&, ||
Indizado (el contador inicia en el elemento 0)	[ ]
Conversiones	( ), as
Asignación	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, ??
Rotación de Bits	<<, >>
Información de Tipos de datos	sizeof, typeof
Concatenación y eliminación de Delegados	+, -
Control de excepción de Overflow	checked, unchecked
Apuntadores y direccionamiento en código No seguro (Unsafe code)	*, ->, [ ], &
Condicional (operador ternario)	?:

Para mayor información acerca de los operadores en Visual C#, se recomienda visitar el siguiente enlace:

C# Operators
https://msdn.microsoft.com/en-us/library/6a71f45d.aspx

Declaración y asignación de variables

Antes de utilizar una variable, debemos declararla. Al declararla podemos especificar su nombre y características. El nombre de la variable es referido como un **Identificador**. Visual C# tiene reglas específicas relacionadas con el uso de los identificadores:

- Un identificador solo puede contener letras, dígitos y el carácter guion bajo.
- Un identificador debe iniciar con una letra o un guion bajo.
- Un identificador no debería ser una de las palabras clave que visual C# reserva para su propio uso.

Visual C# es sensible a mayúsculas y minúsculas, por ejemplo, la variable **IDEmpleado** es distinta a la variable **idempleado**. Nosotros podemos declarar 2 variables llamadas **IDEmpleado** y **idempleado** al mismo tiempo y Visual C# no se confundirá, sin embargo, esta no es una buena práctica de codificación.

Al declarar una variable, debemos elegir un nombre que tenga significado respecto a lo que almacena, de esta forma, el código será más fácil de entender. Debemos también adoptar una convención de nombres y utilizarla.

Microsoft ofrece una nomenclatura que podríamos seguir en caso de no contar con una nomenclatura propia. Para mayor información, se recomienda visitar el siguiente enlace:

Naming Guidelines
https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx

Cuando se declara una variable, se reserva un espacio de almacenamiento en memoria para esa variable y el tipo de datos que va a contener. Podemos declarar múltiples variables en una sola declaración utilizando el separador coma (,), todas las variables declaradas de esta manera, son del mismo tipo de datos. El siguiente ejemplo, muestra como declarar una nueva variable.

```
int Precio;
int Impuesto, Descuento;
```

Después de declarar la variable, podemos asignarle un valor utilizando una operación de asignación. Durante la ejecución de la aplicación, podemos cambiar el valor de una variable tantas veces como queramos. El operador de asignación **=** nos permite asignar un valor a una variable.

El siguiente código muestra cómo utilizar el operador **=** para asignar el valor a una variable.

```
Precio = 120;
```

El valor del lado derecho de la expresión es asignado a la variable que se encuentra en el lado izquierdo de la expresión.

También es posible declarar una variable y asignar su valor al mismo tiempo. El siguiente ejemplo declara una variable de tipo entero llamada **Precio** y le asigna el valor 120.

```
int Precio = 120;
```

Cuando declaramos una variable, esta contiene un valor aleatorio hasta que le asignamos un valor. Este comportamiento fue una poderosa fuente de errores en C y C++ que nos permitían declarar variables y utilizarlas accidentalmente sin haberles asignado previamente un valor. Visual C# no nos permite utilizar una variable que no haya sido previamente asignada. Debemos asignar un valor a una variable antes de que la utilicemos, de lo contrario, la aplicación podría no compilar.

Al declarar las variables, también podemos utilizar la palabra clave **var** en lugar de especificar un tipo de datos explícitamente, como **int** o **string**. Cuando el compilador encuentra la palabra clave **var**, este utiliza el valor que se asigna a la variable para poder determinar el Tipo correspondiente.

En el siguiente ejemplo, se muestra la forma de utilizar la palabra clave **var** para declarar una variable.

```
var Perimetro = 1000;
```

En este caso, la variable **Perimetro** es una variable cuyo tipo fue establecido de forma implícita. Sin embargo, la palabra clave **var** no indica que posteriormente podamos asignarle un valor de un tipo diferente. El tipo de **Perímetro** es fijo de la misma forma que lo sería si lo hubiéramos declarado explícitamente.

Las variables con tipo implícito son útiles cuando no sabemos, o es difícil establecer de forma explícita el tipo de una expresión que se desea asignar a una variable.

Otro tipo común de variables que podemos utilizar, son las variables que almacenan objetos.

Cuando declaramos una variable **objeto**, está se encuentra inicialmente sin asignar. Para utilizar una variable objeto, debemos crear una instancia de la clase correspondiente mediante el operador **new** y asignarla a la variable objeto.

El operador **new** hace dos cosas:

1. Hace que el CLR asigne memoria para el objeto.
2. Después de asignar la memoria para el objeto, invoca un constructor para inicializar los campos de dicho objeto. La versión del constructor que se ejecuta depende de los parámetros que especifiquemos en el operador **new**.

El siguiente ejemplo muestra la forma de crear la instancia de una clase mediante el uso del operador **new**.

```
var Coleccion = new System.Collections.ArrayList();
```

Para mayor información acerca de la declaración y asignación de variables, se recomienda visitar el siguiente enlace:

Implicitly Typed Local Variables (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/bb384061.aspx

Accediendo a los miembros de un Tipo

Para acceder a un miembro de una instancia de un tipo, utilizamos el nombre de la instancia, seguido por un punto, seguido por el nombre del miembro. Esto es conocido como la Notación de Punto (Dot Notation). Cuando accedamos a los miembros de una instancia debemos considerar las siguientes reglas y guías:

- Para acceder a un método, utilizamos paréntesis después del nombre del método. Dentro de los paréntesis, pasamos los valores de cualquier parámetro que el método requiera. Si el método no requiere parámetros, los paréntesis siguen siendo requeridos.
- Para acceder a una propiedad pública, utilizamos el nombre de la propiedad. Podemos obtener el valor de la propiedad o establecer el valor de la propiedad. A diferencia de los métodos, las propiedades no deben llevar paréntesis.

El siguiente ejemplo muestra como invocar los miembros que la clase **Empleado** expone.

```
// Creamos la instancia de la clase Empleado
var E = new Empleado();
// Establecemos el valor de la propiedad SueldoDiario
E.SueldoDiario = 500;
// Obtenemos el valor de la propiedad SueldoDiario
var Sueldo = E.SueldoDiario;
// Invocamos al método CalcularPago
var Pago = E.CalcularPago(15);
// Invocamos el método CalcularPagoMensual
var PagoMensual = E.CalcularPagoMensual();
```

Para mayor información sobre el uso de métodos y propiedades, se recomienda visitar los siguientes enlaces:

Properties (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx

Methods (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/ms173114.aspx

Conversiones entre tipos de datos

Al momento de desarrollar una aplicación, frecuentemente necesitamos convertir datos de un tipo a otro tipo, por ejemplo, cuando un valor de un tipo es asignado a una variable de tipo diferente. Consideremos el escenario donde un usuario proporciona un número en una caja de texto o a través de la ventana de línea de comandos. Para utilizar este número en un cálculo aritmético, necesitaremos convertir el valor texto proporcionado por el usuario hacia un valor entero que podemos almacenar en una variable de tipo entero. El proceso de convertir un valor de un tipo de datos a otro tipo de datos es llamado **Conversión de Tipo** o **Casting**.

Hay 2 tipos de conversión en el .NET Framework:

- **Conversión Implícita**. La conversión implícita es realizada automáticamente por el CLR en las operaciones donde existe la seguridad de que no se perderá la información al momento de realizar la conversión.
- **Conversión Explicita**. La conversión explicita requiere que escribamos código para realizar una conversión que de otra forma podría ocasionar perdida de información o generar un error.

La conversión explicita reduce la posibilidad de errores en nuestro código y hace que el código sea más eficiente. Visual C# prohíbe las conversiones implícitas que pierden precisión. Sin embargo, debemos tener en cuenta que algunas conversiones explicitas pueden originar resultados inesperados, sobre todo cuando el tipo al que queremos convertir no es totalmente compatible con el tipo que tenemos.

El siguiente ejemplo, muestra como el dato es convertido implícitamente desde un **byte** hacia un **int**. Esta conversión es conocida como ***Widening*** (Cuando convertimos un tipo pequeño a un tipo más grande).

```
int i;
byte b = 20;
i = b;
```

La conversión implícita siempre es exitosa y nunca resultará en una pérdida de información. Sin embargo, no podemos convertir implícitamente un tipo **int** a **byte** debido a que la conversión corre el riesgo de perder información (el valor entero podría ser más grande que el valor soportado por el byte). La siguiente tabla muestra los tipos de conversión implícita que son soportados en Visual C#.

Desde	Hacia
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal

ushort	int, uint, long, ulong, float, double, decimal
Int	long, float, double, decimal
Uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
Float	Double
Char	ushort, int, uint, long, ulong, float, double, decimal

En Visual C# podemos utilizar el operador **Cast** para realizar conversiones explicitas. Un **Cast** especifica dentro de paréntesis el tipo hacia el cual queremos convertir.

El siguiente ejemplo muestra cómo realizar una conversión explicita. Esta conversión es conocida como ***Narrowing*** (Cuando convertimos un tipo grande a un tipo más pequeño).

```
byte b;
int i = 20;
b = (byte)i;
```

Únicamente podemos realizar conversiones significativas de esta forma, tal como convertir **int** a **byte**. No podemos utilizar **Cast** si el formato del dato tiene que cambiar físicamente, tal como si quisiéramos convertir un **string** a **int**. Para realizar conversiones de ese tipo, podemos utilizar los métodos de la clase **System.Convert**.

La clase **System.Convert** proporciona métodos que podemos utilizar para convertir de un tipo de dato a otro. Estos métodos tienen nombres tales como **ToDouble**, **ToInt32**, **ToString**, etc. Todos los lenguajes que generan código para el CLR pueden utilizar esta clase. Nos podría parecer que esta clase es más fácil de utilizar para realizar las conversiones en lugar de las conversiones implícitas o explicitas debido a que el IntelliSense nos ayuda a localizar el método de conversión que necesitemos.

El siguiente ejemplo, convierte un **string** a **byte**.

```
string s = "127";
byte n = System.Convert.ToByte(s);
```

Algunos de los tipos de datos predefinidos en Visual C# proporcionan un método **TryParse** que permite determinar cuando la conversión será exitosa antes de realizar la conversión misma.

El siguiente ejemplo muestra cómo convertir un **string** a **byte** mediante el uso del método **byte.TryParse()**.

```
string s = "127";
byte obyte;
bool exito = byte.TryParse(s, out obyte);
```

En el código anterior, la variable **exito** tendrá un valor **True** si la conversión pudo realizarse con éxito y tendrá un valor **False** en caso contrario.

Para mayor información sobre el tema de conversiones de variables, se recomienda visitar el siguiente enlace:

Casting and Type Conversions (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/ms173105.aspx

Manejo de cadenas

Las cadenas son tipos de datos muy útiles que nos permiten capturar y almacenar datos alfanuméricos.

La forma más simple de concatenar cadenas en Visual C# es utilizando el operador "+". Sin embargo, esto es considerado una mala práctica de programación debido a que las cadenas son inmutables. Esto significa que cada vez que concatenamos una cadena, creamos una nueva cadena en memoria y la cadena anterior es eliminada.

El siguiente ejemplo crea 5 cadenas al ejecutarse.

```
string direccion = "13 Oriente ";
direccion = direccion + " No. 1810";
direccion = direccion + "Colonia Centro";
```

La recomendación en este caso es utilizar la clase **StringBuilder** que permite construir una cadena de manera dinámica y mucho más eficiente.

El siguiente código muestra cómo utilizar la clase **StringBuilder**.

```
StringBuilder direccion1 = new StringBuilder();
direccion1.Append("13 Oriente ");
direccion1.Append(" No. 1810");
direccion1.Append("Colonia Centro");
```

Validando cadenas

Cuando solicitamos datos a través de la interfaz de usuario de una aplicación, los datos frecuentemente son proporcionados como cadenas de texto que necesitamos validar y posteriormente convertir en un formato que la lógica de la aplicación espera. Por ejemplo, un control **TextBox** en una aplicación WPF, devolverá su contenido como una cadena, aunque el usuario haya proporcionado solo caracteres numéricos. Es importante que nosotros validemos ese tipo de entrada para minimizar el riesgo de errores, tales como **InvalidCastExceptions**.

Las expresiones regulares proporcionan un mecanismo que nos permite validar la entrada de datos. El NET Framework proporciona el espacio de nombres System.Text.RegularExpressions que incluye la clase **Regex**. Nosotros podemos utilizar la clase **Regex** en nuestras aplicaciones para probar un **string** y así asegurarnos de que cumple con las condiciones de una expresión regular.

El siguiente código muestra la forma de utilizar el método **Regex.IsMatch** para verificar si el valor de un **string** contiene dígitos numéricos.

```
var Texto = "Hello Word";
var ExpressionRegular = @"\d";
var Resultado = Regex.IsMatch(Texto, ExpressionRegular, RegexOptions.None);
if (Resultado)
{
    Console.WriteLine("Contiene digitos");
}
else
{
    Console.WriteLine("No contiene digitos");
}
```

Las expresiones regulares proporcionan una selección de expresiones que podemos utilizar para hacer coincidir una gran variedad de tipos de datos. Por ejemplo, la expresión **\d** coincidirá con cualquier carácter numérico.

Para mayor información sobre el tema de Expresiones Regulares, se recomienda visitar los siguientes enlaces:

Regex Class
https://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex(v=vs.110).aspx

Regular Expression Language - Quick Reference
https://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx

Lección 3:
Sentencias principales del lenguaje de programación C#

Cuando desarrollamos una aplicación, frecuentemente necesitamos ejecutar lógica basada en una condición, o ejecutar repetidamente una sección de código hasta que se cumpla una condición. También es posible que necesitemos almacenar una colección de datos relacionados en una sola variable. Visual C# proporciona diversos elementos que nos permiten resolver tareas complejas en nuestras aplicaciones.

En esta lección, conoceremos la forma de implementar instrucciones de decisión e iteración, así como la forma de almacenar colecciones de datos relacionados. Aprenderemos también a estructurar las clases de nuestras aplicaciones mediante el uso de espacios de nombres. Finalmente, conoceremos algunas de las características de depuración que Visual Studio proporciona.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Utilizar sentencias condicionales.
- Utilizar sentencias de iteración.
- Crear y utilizar arreglos.
- Describir el propósito de los espacios de nombres.
- Utilizar puntos de ruptura en Visual Studio.

Implementando lógica condicional

La lógica de una aplicación frecuentemente necesita ejecutar diferentes secciones de código dependiendo del estado de los datos en la aplicación. Por ejemplo, si un usuario solicita cerrar un archivo, se le podría preguntar si desea guardar sus cambios. Si el usuario acepta guardar los cambios, la aplicación debe ejecutar código para guardar el archivo. Si el usuario no acepta guardar sus cambios, la lógica de la aplicación simplemente puede cerrar el archivo. Visual C# utiliza sentencias condicionales para determinar cuál sección de código ejecutar.

La sentencia condicional primaria en Visual C# es la sentencia **if**. Existe también la sentencia Switch que podemos utilizar para decisiones más complejas.

Sentencias Condicionales

Utilizamos sentencias **if** para probar la veracidad de una expresión. Si la expresión es **true**, el bloque de código asociado con la sentencia **if** es ejecutado, si la expresión es **false**, el control continúa fuera del bloque **if**.

El siguiente código muestra la forma de utilizar una instrucción **if** para verificar que el usuario haya escrito un número desde la Consola.

```
static void Main()
{
    Console.Write("Escribe un número: ");
    var Respuesta = Console.ReadLine();
    int Numero;
    if (int.TryParse(Respuesta, out Numero))
    {
        Console.WriteLine($"Has proporcionado el número {Numero}");
    }
}
```

Las sentencias **if** pueden tener clausulas **else** asociadas. El bloque **else** se ejecuta cuando la sentencia **if** es **false**.

El siguiente ejemplo, muestra cómo utilizar una instrucción **if else** para ejecutar código cuando una condición sea evaluada como **false**.

```
if (int.TryParse(Respuesta, out Numero))
{
    Console.WriteLine($"Has proporcionado el número {Numero}");
}
else
{
    Console.WriteLine("Debes proporcionar un número entero");
}
```

Las sentencias **if** también pueden tener asociadas sentencias **else if**. Estas sentencias son evaluadas en el orden en el que aparecen en el código después de la sentencia **if**. Si una de las clausulas devuelve **true**, el bloque de código asociado con esa cláusula es ejecutado y la ejecución deja el bloque entero **if**.

El siguiente ejemplo muestra cómo utilizar una instrucción **if** con una clausula **else if**, para permitir al usuario proporcionar un número entero o bien, la palabra **FIN**.

```
if (int.TryParse(Respuesta, out Numero))
{
    Console.WriteLine($"Has proporcionado el número {Numero}");
}
else if(Respuesta == "FIN")
{
    Console.WriteLine("Has escrito la palabra FIN");
}
else
{
    Console.WriteLine("Debes proporcionar un número entero");
}
```

Instrucciones de Selección

En caso de que existan demasiadas instrucciones **if/else**, el código se puede volver difícil de entender. En este caso, una solución mejor es utilizar la instrucción **switch** para remplazar las múltiples sentencias **if/else**.

El siguiente ejemplo muestra cómo utilizar la instrucción **switch** para remplazar una colección de cláusulas **if/else**.

```
switch(Numero)
{
    case 1:
        Console.WriteLine("Has seleccionado la opción Altas.");
        break;
    case 2:
        Console.WriteLine("Has seleccionado la opción Bajas.");
        break;
    case 3:
        Console.WriteLine("Has seleccionado la opción Consultas.");
        break;
    default:
        Console.WriteLine("Debes seleccionar una opción correcta.");
        break;
}
```

Notemos que en cada sentencia **case**, tenemos una sentencia **break**. Esto causa que el control salte hacia el final del **switch** después de procesar el bloque de código. Si omitimos la palabra clave **break** el código no compilará.

Notemos también que existe un bloque llamado **default**, este bloque de código será ejecutado cuando ninguno de los bloques **case** cumpla la condición.

Para mayor información sobre las sentencias de Selección, se recomienda visitar el siguiente enlace:

Selection Statements (C# Reference)
https://msdn.microsoft.com/en-us/library/676s4xab(v=vs.140).aspx

Implementando lógica de Iteración

La iteración proporciona una manera conveniente de ejecutar un bloque de código múltiples veces. Por ejemplo, iterando sobre una colección de elementos en un arreglo o simplemente ejecutando una función múltiples veces.

Visual C# proporciona un conjunto de sentencias estándar conocidas como ciclos que podemos utilizar para implementar la lógica de una iteración.

Ciclos For

Los ciclos **For**, ejecutan un código repetidamente hasta que la expresión especificada se evalúe en **false**. Podemos definir un ciclo **for** de la siguiente manera:

```
for ([Inicializadores]; [Expresión a evaluar]; [Iteradores])
{
  [Cuerpo]
}
```

Normalmente, cuando utilizamos un ciclo **for**, primero inicializamos un valor como un contador. En cada iteración del ciclo, verificamos que el valor del contador este dentro del rango para ejecutar el ciclo **for** y si es así, ejecutamos el cuerpo del ciclo.

El siguiente código muestra cómo utilizar un ciclo **for** para ejecutar un bloque de código 10 veces.

```
for (int i = 0; i < 10; i++)
{
    // Código a ejecutar.
}
```

En este ejemplo, **i = 0** es el inicializador, **i < 10** es la expresión que se evalúa e **i++** es el iterador.

Ciclos For Each

Aunque un ciclo **for** es fácil de utilizar, no en todos los casos podría ser la mejor solución. Por ejemplo, cuando iteramos sobre una colección o un arreglo, necesitamos conocer cuántos elementos contiene la colección o arreglo. En muchos casos esto es muy sencillo, pero en algunas ocasiones podríamos equivocarnos fácilmente. Por lo tanto, algunas veces es mejor utilizar un ciclo **foreach**.

El ciclo **foreach**, nos permite recorrer los elementos de una colección sin necesidad de conocer cuántos elementos tiene dicha colección.

El siguiente ejemplo muestra cómo utilizar el ciclo **foreach** para iterar un arreglo de cadenas.

```
string[] Cadenas = new string[10];
// Procesar cada cadena en el arreglo
foreach (string Cadena in Cadenas)
{
    // Código a ejecutar
}
```

Ciclos While

Un ciclo **While** nos permite ejecutar un bloque de código mientras una determinada condición sea **true**. Por ejemplo, podemos utilizar un ciclo **While** para procesar la entrada del usuario hasta que el usuario indique que ya no tiene más datos para proporcionar.

El siguiente código muestra cómo utilizar el ciclo While.

```
string Dato = Console.ReadLine();
while (Dato!="")
{
    // Procesar el dato

    Dato = Console.ReadLine();
}
```

Ciclos Do

Un ciclo **do** es similar al ciclo **while** con la excepción de que el ciclo **do** siempre será ejecutado al menos una vez a diferencia del ciclo **while** donde si la condición inicial no se cumple, el ciclo nunca se ejecutará. El ciclo **do** primero ejecuta el cuerpo del ciclo y después compara la condición. Por ejemplo, podemos utilizar el ciclo do si sabemos que el código solo será ejecutado en respuesta a una solicitud del usuario para proporcionar datos. En este escenario, sabemos que la aplicación necesitará procesar al menos una pieza de datos y, en consecuencia, podemos utilizar un ciclo **do**.

El siguiente ejemplo muestra el uso del ciclo **do**.

```
do
{
    Dato = Console.ReadLine();
    // Procesar el dato

} while (Dato != "");
```

Para mayor información sobre las sentencias de Ciclos, se recomienda visitar el siguiente enlace:

Iteration Statements (C# Reference)
https://msdn.microsoft.com/en-us/library/32dbftby.aspx

Creando y utilizando Arreglos

Un arreglo es un conjunto de objetos agrupados y manejados como una unidad. Podemos pensar de un arreglo como una secuencia de elementos, todos del mismo tipo.

Podemos construir arreglos simples que tengan una sola dimensión (una lista), dos dimensiones (una tabla), tres dimensiones (un cubo) y así sucesivamente. Los arreglos en C# tienen las siguientes características:

- Cada elemento en el arreglo contiene un valor.
- Los arreglos son **zero-indexed**, esto significa que el primer elemento del arreglo es 0.
- El tamaño de un arreglo es el número total de elementos que contiene.
- Los arreglos pueden ser de una sola dimensión, multidimensionales o jagged.
- El rango de un arreglo es el número de dimensiones del arreglo.

Los arreglos de un tipo particular solo pueden contener elementos de ese tipo. Si necesitamos manipular un conjunto de elementos de tipos distintos, debemos considerar el uso de una colección de tipos definidos en el espacio de nombre **System.Collections**.

Crear Arreglos

Cuando declaramos un arreglo, especificamos el tipo de dato que contendrá y el nombre del arreglo. Al declarar un arreglo, hacemos que el arreglo sea accesible pero no se localiza memoria para él. El CLR crea físicamente el arreglo cuando utilizamos la palabra clave **new**. En ese punto deberíamos especificar el tamaño del arreglo.

La siguiente lista describe como crear arreglos de una sola dimensión, multidimensionales o **jagged**.

- **Arreglos de una sola dimensión**. Para declarar un arreglo de una sola dimensión, especificamos el tipo de elementos en el arreglo y utilizamos corchetes cuadrados **[ ]** para indicar que una variable es un arreglo. Posteriormente, especificamos el tamaño del arreglo cuando obtenemos memoria para el arreglo mediante el uso de la palabra clave **new**. El tamaño de un arreglo puede ser cualquier expresión entera. El siguiente ejemplo muestra cómo crear un arreglo de enteros de una sola dimensión con los elementos del 0 al 9.

```
int[] ArregloDeEnteros = new int[10];
```

- **Arreglos multidimensionales**. Un arreglo puede tener más de una dimensión. El número de dimensiones de un arreglo corresponde al número de índices que son utilizados para identificar a un elemento individual del arreglo. Podemos especificar hasta 32 dimensiones, pero rara vez necesitaremos más de 3. Declaramos una variable arreglo multidimensional de la misma forma en que declaramos un arreglo de una sola dimensión, pero separamos las

dimensiones mediante comas. El siguiente ejemplo muestra cómo crear un arreglo de enteros con 3 dimensiones.

```
int[,,] ArregloDeEnteros = new int[10,10,10];
```

- **Arreglos Jagged**. Un arreglo Jagged es simplemente un arreglo de arreglos donde el tamaño de cada arreglo puede variar. Los arreglos **Jagged** son útiles para modelar estructuras de datos dispersas donde no siempre quisiéramos asignar memoria a cada elemento si este no va a ser utilizado. El siguiente ejemplo muestra como declarar e inicializar un arreglo Jagged. Nota que debemos especificar el tamaño del primer arreglo, pero no debemos especificar el tamaño de los arreglos que estarán contenidos dentro de ese arreglo. Asignamos memoria para cada arreglo dentro del arreglo Jagged de forma separada, utilizando la palabra clave new.

```
int[][] JaggedArray = new int[10][];
JaggedArray[0] = new int[5]; // Podemos especificar diferentes tamaños
JaggedArray[1] = new int[7];

//.....

JaggedArray[9] = new int[21];
```

Accediendo a los datos de un arreglo

Podemos acceder a los elementos de un arreglo de distintas formas, tal como especificando el índice de un elemento especifico que sea requerido o iterando a tarves de la colección entera y devolviendo cada elemento en secuencia.

El siguiente ejemplo utiliza un índice para acceder al elemento con el índice 2.

```
int[] NumerosNones = { 1, 2, 3, 4, 5 };
int Numero = NumerosNones[2];
```

Los arreglos son Zero-indexed, de tal forma que el primer elemento en cualquier dimensión en un arreglo se encuentra en el índice 0. El último elemento en una dimensión se encuentra en el índice N-1, donde N es el tamaño de la dimensión. Si tratamos de acceder a un elemento fuera de este rango, el CLR lanza una excepción **IndexOutOfRangeException**.

Podemos iterar a través de un arreglo utilizando un ciclo **for**. Podemos utilizar la propiedad **Length** del arreglo para determinar cuándo debemos detener el ciclo.

El siguiente ejemplo muestra cómo utilizar el ciclo **for** para iterar a través del arreglo.

```
int[] NumerosNones = { 1, 2, 3, 4, 5 };
for(int i = 0; i < NumerosNones.Length; i++)
{
    int Numero = NumerosNones[i];

    // ....
}
```

Para mayor información sobre el manejo de arreglos, se recomienda visitar el siguiente enlace:

Arrays (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/9b9dty7d.aspx

Referenciando espacios de nombre

La Biblioteca de Clases Base del Microsoft .NET Framework consiste de muchos espacios de nombres (**Namespaces**) que organizan sus clases en jerarquías relacionados lógicamente. Nosotros podemos utilizar espacios de nombres en nuestras propias aplicaciones para organizar nuestras clases de manera similar en jerarquías.

Los Espacios de nombres funcionan tanto como un sistema interno para organizar nuestra aplicación como de forma externa para evitar conflictos de nombres entre nuestro código y el de otras aplicaciones. Cada espacio de nombres contiene tipos que podemos utilizar en nuestro programa, tales como clases, estructuras, enumeraciones, delegados e interfaces. Debido a que diferentes clases pueden tener el mismo nombre, utilizamos espacios de nombres para diferenciar el mismo nombre de clase en dos diferentes jerarquías y evitar conflictos de interoperabilidad.

Espacios de nombres de la Biblioteca de Clases Base de .NET Framework

El espacio de nombre más importante del .NET Framework es el espacio de nombres **System**. Este espacio de nombres contiene clases que la mayoría de las aplicaciones utilizan para interactuar con el sistema operativo. La siguiente tabla lista algunos de los espacios de nombres proporcionados por el .NET Framework a través del espacio de nombres **System**.

Namespace	Definición
System.Windows	Proporciona las clases que son útiles para crear aplicaciones WPF.
System.IO	Proporciona clases para leer y escribir datos a archivos.
System.Data	Proporciona clases para acceso a datos.
System.Web	Proporciona clases que son útiles para desarrollar aplicaciones Web.

Espacios de nombres definidos por el usuario

Es una buena práctica definir todas nuestras clases dentro de espacios de nombres definidos en nuestro código. El ambiente de Visual Studio sigue estas recomendaciones utilizando el nombre de nuestro proyecto como el espacio de nombres principal del proyecto.

El siguiente ejemplo muestra cómo definir un espacio de nombres con el nombre **TICapacitacion.Console** que contiene la clase **Program**.

```
namespace TICapacitacion.Console
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Utilizando espacios de nombres

Cuando creamos un proyecto Visual C# en Visual Studio, los Assemblies con las clases base más comunes son referenciados. Sin embargo, si necesitamos utilizar un tipo que se encuentra en un Assembly que no se encuentra referenciado actualmente, necesitaremos agregar una referencia al Assembly mediante la caja de dialogo **Add Reference**. Posteriormente, al inicio de nuestro archivo de código, podemos listar los espacios de nombres que utilicemos en ese archivo de código precediéndolos por la directiva **using**. La directiva using es un acceso directo que indica a la aplicación que los tipos en el espacio de nombres pueden ser referenciados directamente, sin utilizar el nombre completamente calificado del tipo que incluye el espacio de nombres y el nombre del tipo.

El siguiente ejemplo, muestra como importar el espacio de nombres **System** y utilizar la clase **Console**.

```
using System;
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

El siguiente ejemplo, es similar al anterior, pero sin importar el espacio de nombres **System**. Esto nos obliga a utilizar el nombre completamente calificado del tipo **Console**.

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello, World!");
        }
    }
}
```

Para mayor información sobre los espacios de nombres, se recomienda visitar el siguiente enlace:

namespace (C# Reference)
https://msdn.microsoft.com/en-us/library/z2kcy19k(v=vs.140).aspx

Utilizando puntos de ruptura en Visual Studio

La depuración es una parte esencial del desarrollo de aplicaciones. Podemos notar los errores al escribir el código, pero algunos errores, especialmente errores de lógica, pueden ocurrir únicamente en circunstancias que no predecimos. Los usuarios podrían informarnos de estos errores y tendríamos que corregirlos.

Visual Studio proporciona varias herramientas para ayudarnos a depurar el código. Podemos utilizar estas herramientas mientras desarrollamos el Código, durante una fase de prueba o después de que la aplicación haya sido liberada. Utilizamos las herramientas de la misma manera sin importar las circunstancias. Podemos ejecutar una aplicación con o sin depuración habilitada. Cuando la depuración está habilitada, se dice que la aplicación está en modo Debug.

Utilizando puntos de ruptura

Si conocemos la ubicación aproximada del problema en el código, podemos utilizar un punto de interrupción para que el depurador de Visual Studio acceda al modo de interrupción antes de la ejecución de una línea de código específica. Esto nos permite utilizar las herramientas de depuración para revisar o modificar el estado de nuestra aplicación para ayudarnos a rectificar el error. Para añadir un punto de interrupción a una línea de código, en el menú **Debug**, hacemos clic en **Toggle Breakpoint**.

Cuando nos encontramos en modo de interrupción, podemos pasar el mouse sobre los nombres de las variables para ver su valor actual. También podemos utilizar la ventana **Immediate Window** y los paneles **Autos**, **Locals** y **Watch** para ver y modificar el contenido de las variables.

Utilizando los controles de depuración

Después de ver o modificar las variables en el modo de interrupción, es probable que deseemos desplazarnos sobre las siguientes líneas del código de la aplicación. Podemos ejecutar simplemente el resto de la aplicación o podemos ejecutar una línea de código a la vez. Visual Studio proporciona una variedad de comandos en el menú **Debug** que nos permiten hacer esto y mucho más.

La siguiente tabla lista los controles principales de depuración.

Opción del Menú	Botón en la barra de herramientas	Combinación de teclas	Descripción
Iniciar Depuracion (Start Debugging)	Iniciar/continuar	F5	Este botón está disponible cuando la aplicación no está en ejecución y cuando se encuentra en modo de interrupción. Esta opción inicia la aplicación en modo de depuración o reanuda la aplicación si se

			encuentra en modo de interrupción.
Interrumpir Todo (Break All)	Break all	Ctrl+Alt+Break	Este botón causa que la ejecución de la aplicación entre en modo de depuración. El botón está disponible cuando la aplicación se encuentra en modo de ejecución.
Detener la Depuración (Stop Debugging)	Stop	Shift+F5	Este botón detiene la depuración. Está disponible cuando una aplicación se está ejecutando o está en modo de interrupción.
Reanudar (Restart)	Restart	Ctrl+Shift+F5	Este botón es equivalente a detener e iniciar. Causa que la aplicación se reinicie desde el principio. Está disponible cuando la aplicación se está ejecutando o está en modo de interrupción.
Entrar en (Step Into)	Step into	F11	Este botón se utiliza para entrar al código de los métodos cuando se realiza una llamada a ellos.
Pasar por encima (Step Over)	Step over	F10	Este botón se utiliza para no entrar al código de los métodos cuando se realiza una llamada a ellos.
Ejecutar y salir Step Out	Step out	Shift+F11	Este botón se utiliza para ejecutar el código restante en el método y regresa a la siguiente instrucción en el método que ha invocado.

Para mayor información sobre la depuración en Visual Studio, se recomienda visitar el siguiente enlace:

Debugging in Visual Studio
https://msdn.microsoft.com/en-us/library/sc65sadd(v=vs.140).aspx

Introducción a C#

Módulo 2: Creación de Métodos, Manejo de Excepciones y Monitoreo de Aplicaciones

Acerca del módulo

Las aplicaciones consisten frecuentemente de unidades lógicas de funcionalidad que realizan funciones específicas, tales como proporcionar acceso a datos o lanzar algún procesamiento lógico. Visual C# es un lenguaje orientado a objetos y utiliza el concepto de métodos para encapsular unidades lógicas de funcionalidad. Un método puede ser tan simple o complejo como queramos, por lo tanto, es importante considerar los cambios en el estado de la aplicación que suceden cuando una excepción es generada en un método.

En este módulo, aprenderemos a crear y utilizar métodos, así como la forma de manejar excepciones. Conoceremos algunas técnicas de seguimiento que nos permitan registrar el detalle de las excepciones que ocurran.

Objetivos

Al finalizar este módulo, los participantes contarán con las habilidades y conocimientos para:

- Crear e invocar métodos.
- Crear métodos sobrecargados
- Utilizar parámetros opcionales y argumentos nombrados.
- Manejar excepciones.
- Monitorear aplicaciones mediante el uso de las características de ***logging***, ***tracing*** y ***profiling***.

Los temas que se cubren en este módulo son:

- Lección 1: Creando e invocando Métodos.
- Lección 2: Creando Métodos sobrecargados y utilizando parámetros opcionales y de salida (output).
- Lección 3: Manejo de Excepciones.
- Lección 4: Monitoreo de aplicaciones.

Lección 1:
Creando e invocando Métodos

Una parte clave en el desarrollo de cualquier aplicación es dividir la solución en componentes lógicos. En los lenguajes orientados a objetos, tales como Visual C#, un método es una unidad de código que realiza una pieza discreta de trabajo.

En esta lección, aprenderemos a crear e invocar métodos.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Describir el propósito de los métodos.
- Crear métodos.
- Invocar métodos.
- Depurar métodos.

¿Qué es un método?

La habilidad de definir e invocar métodos, es un componente fundamental de la programación orientada a objetos debido a que los métodos permiten encapsular operaciones que protegen los datos almacenados dentro de un tipo.

Normalmente, cualquier aplicación que desarrollemos utilizando el Microsoft .NET Framework y Visual C#, tendrá varios métodos, cada uno con un propósito específico. Algunos métodos son fundamentales para la operación de una aplicación. Por ejemplo, todas las aplicaciones de escritorio de Visual C#, deben tener un método llamado **Main** que define el punto de entrada de la aplicación. Cuando el usuario ejecuta una aplicación Visual C#, el CLR ejecuta el método **Main** de esa aplicación.

Los métodos pueden ser diseñados para uso interno de un tipo y como tal, se encuentran ocultos para otros tipos. Los métodos públicos son expuestos fuera del tipo y pueden ser diseñados para permitir a otros tipos solicitar que un objeto realice una acción.

El mismo .NET Framework está compuesto por clases que exponen métodos que podemos invocar desde nuestras aplicaciones para interactuar con el usuario y la computadora.

Creando Métodos

Un método está compuesto por dos elementos:

1. La especificación del método.
2. El cuerpo del método.

La especificación del método define el nombre del método, los parámetros que el método puede tomar, el tipo de valor devuelto del método y los modificadores de acceso del método.

La combinación del nombre del método y su lista de parámetros se conoce como ***la firma del método***.

La definición del valor devuelto de un método **no** se considera como parte de la firma. Cada método en una clase debe tener una firma única.

Nombrando un método

Un nombre de método tiene las mismas restricciones sintácticas que el nombre de una variable. El nombre del método debe comenzar con una letra o un guion bajo y sólo puede contener letras, guiones y caracteres numéricos. Visual C# es sensible a mayúsculas y minúsculas, por lo que una clase puede contener dos métodos que tengan el mismo nombre y se diferencien sólo en el uso de mayúsculas y minúsculas de una o más letras, aunque esto no es una buena práctica de codificación.

Dos lineamientos recomendados por Microsoft como una buena práctica al elegir el nombre del método son:

1. Utilizar verbos o frases de verbos para nombrar un método. Esto ayuda a que otros desarrolladores puedan entender la estructura de nuestro código.
2. Utilizar la nomenclatura *Pascal Casing*. No se recomienda iniciar los nombres de los métodos con un guion o una letra minúscula.

Implementando un cuerpo de Método

El cuerpo de un método es un bloque de código que se implementa mediante el uso de cualquiera de las sentencias de programación de Visual C#. El cuerpo está encerrado entre llaves.

Podemos definir variables dentro del cuerpo de un método, en cuyo caso solo existirán mientras se esté ejecutando el método. Cuando el método finalice, las variables declaradas ya no serán accesibles.

El siguiente ejemplo muestra el código del método **ShowMessage** que contiene una variable llamada **Status**. La variable **Status** solo está disponible dentro del bloque de código **ShowMessage**. Si

tratamos de utilizar la variable **Status** fuera del alcance del método, el compilador mostrará un mensaje de error indicando que '**Status'** no existe en el contexto actual.

```
void ShowMessage()
{
    var Status = Service.Status;
    // ......
}
```

Especificando parámetros

Los parámetros son variables locales que se crean cuando el método se ejecuta y les son asignados los valores que se especifican cuando el método es invocado. Los parámetros deben ser especificados entre paréntesis después del nombre del método. Cada parámetro está separado por una coma. Si un método no requiere algún parámetro, se especifica una lista de parámetros vacía (solo se incluyen los paréntesis después del nombre del método). El método **ShowMessage** descrito anteriormente es un ejemplo de un método que no requiere parámetros.

Para cada parámetro, se debe especificar el tipo y el nombre. Por convención, los parámetros utilizan la nomenclatura *Camel Casing*.

Al definir los parámetros que el método acepta, podemos utilizar la palabra clave **ref** para indicar al CLR que debe pasar una referencia del parámetro y no solo el valor del parámetro. El código que proporciona el parámetro por referencia debe inicializarlo previamente. Cualquier cambio al valor del parámetro dentro del cuerpo del método se verá reflejado en la variable proporcionada como parámetro al invocar el método.

El siguiente ejemplo muestra cómo definir un parámetro utilizando la palabra clave **ref**.

```
void SaveItems(ref int savedItems)
{
    savedItems = Service.ItemsCount;
    // ....
}
```

Para obtener más información acerca de la palabra clave **ref**, se recomienda visitar el siguiente enlace:

ref (C# Reference)
https://msdn.microsoft.com/en-us/library/14akc2c7.aspx

Especificando un valor de Retorno

Todos los métodos deben tener un tipo de retorno. Un método que no devuelve un valor tiene el tipo de retorno **void**. Al definir el método, especificamos el tipo devuelto antes del nombre del método. Cuando se declara un método que devuelve datos, debemos incluir la instrucción **return** en el cuerpo del método.

El siguiente ejemplo muestra como devolver un tipo **string** desde un método.

```
string GetServiceErrorMessage()
{
    return "El servicio no se encuentra disponible.";
}
```

La expresión que especifica la sentencia **return**, debe tener el mismo tipo que el método declara devolver. Cuando la sentencia **return** se ejecuta, la expresión es evaluada y devuelta a la instrucción que invocó al método. Después de la sentencia **return**, el método finaliza y las instrucciones que pudieran estar después de la sentencia **return** no serán ejecutadas.

Invocando Métodos

Para ejecutar el código de un método, es necesario invocar a ese método. No es necesario entender cómo funciona el código en ese método. Incluso, pudiéramos no tener acceso al código si el método se encuentra en una clase de un ensamblado del que no tenemos el código fuente, como en el caso de la biblioteca de clases base del .NET Framework.

Para invocar a un método, especificamos el nombre del método y entre paréntesis, proporcionamos los argumentos que corresponden a los parámetros del método.

El siguiente ejemplo muestra como invocar el método **RunProcess** pasándole las variables **int** y **bool** para satisfacer los requerimientos de parámetros.

```
void DoJobs()
{
    int Process = 1;
    bool Stop = true;

    RunProcess(Process, Stop);
}
void RunProcess(int numProcess, bool stopIfError)
{
    // .....
}
```

Si el método devuelve un valor, en el código de llamada debemos especificar cómo manejar este valor, por lo general asignándolo a una variable del mismo tipo.

El siguiente ejemplo muestra como capturar el valor de retorno del método **GetServiceErrorMessage** en una variable llamada **Message**.

```
void DoJobs()
{
    string Message = GetServiceErrorMessage();
}


string GetServiceErrorMessage()
{
    return "El servicio no se encuentra disponible.";
}
```

Para invocar a un método que requiera parámetros **ref**, en el código que invoca debemos declarar una variable e inicializarla antes de pasarla como argumento.

El siguiente ejemplo muestra como invocar el método **SaveItems** pasándole una variable **int** por referencia para satisfacer los requerimientos de parámetros.

```
void DoJobs()
{
    int Result=0;
    SaveItems(ref Result);
}
void SaveItems(ref int savedItems)
{
    savedItems = Service.ItemsCount;
    // ....
}
```

En el ejemplo anterior, podemos notar que estamos inicializando la variable **Result** antes de pasarla como parámetro, de no hacerlo, obtendremos un mensaje de error **Use of unassigned local variable 'Result'**.

Para obtener más información acerca de los métodos en Visual C#, se recomienda visitar el siguiente enlace:

Methods (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/ms173114.aspx

Depurando Métodos

Cuando depuramos una aplicación, podemos ejecutar el código instrucción por instrucción, una instrucción a la vez. Esta es una característica extremadamente útil debido a que nos permite probar paso a paso la lógica que la aplicación utiliza.

Visual Studio nos proporciona una serie de herramientas de depuración que nos permiten ejecutar cada línea de código en la manera exacta que lo deseamos. Por ejemplo, podemos avanzar línea por línea de código en cada método que sea ejecutado o podemos ejecutar un bloque de código sin recorrerlo línea por línea.

Al depurar métodos, utilizamos 3 principales características de depuración:

- **Step into**. La característica **Step Into** ejecuta la instrucción en la posición actual de ejecución. Si la declaración es una llamada de método, la posición de ejecución actual se moverá al código dentro del método. Después de haber entrado en un método, se puede continuar la ejecución de instrucciones dentro del método, una línea a la vez. También podemos utilizar la característica **Step Into** para iniciar una aplicación en modo de depuración. Si realizamos esto, la aplicación entrará en el modo de interrupción tan pronto como se inicie.

- **Step over**. La característica **Step Over** ejecuta la instrucción en la posición actual de ejecución. Sin embargo, con esta característica, no entramos en el código dentro de un método. En vez de eso, el código dentro del método es ejecutado y la posición de ejecución se traslada a la instrucción siguiente a la que invoca al método. La excepción a esto es cuando el código del método contiene un punto de interrupción. En este caso, la ejecución continuará y en el punto de interrupción dentro del método, se detendrá.

- **Step out**. La característica **Step Out** permite ejecutar el resto del código del método. La ejecución continuará hasta regresar a la línea que invoco el método y se detendrá en ese punto.

Para obtener más información acerca de las características de depuración disponibles en Visual Studio, se recomienda visitar el siguiente enlace:

Navigating through Code with the Debugger
https://msdn.microsoft.com/en-us/library/y740d9d3.aspx

Lección 2:
Creando Métodos sobrecargados y utilizando parámetros opcionales y de salida (output)

Hemos visto que podemos definir un método que acepte un número fijo de parámetros. Sin embargo, algunas veces podríamos necesitar escribir algún método que requiera diferentes conjuntos de parámetros dependiendo del contexto en el cual sea utilizado. Podemos crear métodos sobrecargados con firma únicas para satisfacer esta necesidad.

En otros escenarios, podríamos querer definir un método que tenga un número fijo de parámetros, pero permitir a una aplicación especificar argumentos únicamente para los parámetros que necesite. Podemos hacer esto mediante la definición de un método que tenga parámetros opcionales y entonces utilizar argumentos nombrados para proporcionar el valor de los parámetros por su nombre.

En esta lección, aprenderemos a crear métodos sobrecargados, definir y utilizar parámetros opcionales, argumentos nombrados, y parámetros de salida (output).

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Crear un método sobrecargado.
- Utilizar parámetros opcionales.
- Utilizar argumentos nombrados.
- Definir parámetros de salida (output).

Creando métodos sobrecargados

Cuando definimos un método, es posible que nos demos cuenta de que nuestro método requiere diferentes conjuntos de información en diferentes circunstancias. Podemos definir métodos sobrecargados para crear múltiples métodos con la misma funcionalidad que acepten diferentes parámetros dependiendo del contexto en el cual sean invocados.

Los métodos sobrecargados tienen el mismo nombre para enfatizar su intención común. Sin embargo, cada método sobrecargado debe tener una firma única para diferenciarlo de otras versiones sobrecargadas del método en la clase.

Recordemos que la firma de un método incluye su nombre y su lista de parámetros. El tipo devuelto no forma parte de la firma. Por lo tanto, no podemos definir métodos sobrecargados que difieran únicamente por el tipo de valor que devuelven.

El siguiente ejemplo, muestra 3 versiones del método **ShowMessage**, todos con firmas únicas.

```
void ShowMessage()
{
    //......
}

void ShowMessage(string message)
{
    // ......
}

void ShowMessage(int messageID)
{
    // ......
}
```

Cuando invocamos al método **ShowMessage**, tenemos que elegir la versión sobrecargada que deseamos utilizar. Para hacer esto, simplemente proporcionamos los argumentos que satisfagan una sobrecarga en particular. De esta forma, el compilador resolverá la versión a invocar basándose en los argumentos que hemos proporcionado.

Creando métodos que utilicen parámetros opcionales

Mediante la definición de métodos sobrecargados, podemos implementar diferentes versiones de un método que requiera diferentes parámetros. Cuando creamos una aplicación que utiliza métodos sobrecargados, el compilador determina qué instancia específica de cada método se debe utilizar para satisfacer cada llamada al método.

Existen otros lenguajes y tecnologías que los desarrolladores pueden utilizar para crear aplicaciones y componentes que no soportan la sobrecarga de métodos. Una característica clave de Visual C# es la capacidad de interoperar con aplicaciones y componentes creados con otras tecnologías. Una de las principales tecnologías que utiliza Windows es el Component Object Model (COM). COM no soporta métodos sobrecargados, en vez de eso, utiliza métodos que pueden tener parámetros opcionales. Para facilitar la incorporación de bibliotecas y componentes COM en una solución de Visual C#, Visual C# proporciona soporte a parámetros opcionales.

Los parámetros opcionales también son útiles en otras situaciones. Proporcionan una solución compacta y simple cuando no es posible utilizar sobrecarga debido a que los tipos de parámetros no varían lo suficiente para permitir al compilador distinguir entre cada implementación.

El siguiente ejemplo, muestra cómo definir un método que acepte un parámetro obligatorio y dos parámetros opcionales.

```
void ShowMessage(string message, string title = null, int iconID = 1)
{
    // ......
}
```

Cuando definimos un método que acepte parámetros opcionales, debemos especificar todos los parámetros obligatorios antes de cualquier parámetro opcional.

El siguiente ejemplo, muestra la definición de un método que utiliza parámetros opcionales que generan un error de compilación.

```
void ShowMessage(string title = null, string message, int iconID = 1)
{
    // ......
}
```

Podemos invocar a un método que acepte parámetros opcionales de la misma manera que invocamos a cualquier otro método. Especificamos el nombre del método y proporcionamos los argumentos necesarios. La diferencia con los métodos que aceptan parámetros opcionales es que podemos omitir los argumentos correspondientes y el método utilizará los valores predeterminados cuando sea ejecutado.

El siguiente ejemplo, muestra como invocar al método **ShowMessage** proporcionando únicamente el argumento obligatorio **message**.

```
ShowMessage("hello, World!");
```

El siguiente ejemplo, muestra como invocar al método **ShowMessage**, proporcionando un argumento para el parámetro obligatorio **message** y un argumento para el parámetro opcional **title**.

```
ShowMessage("hello, World!", "Welcome");
```

Invocando a métodos utilizando argumentos nombrados

Tradicionalmente, cuando invocamos a un método, el orden y la posición de los argumentos en la llamada al método, corresponden al orden de los parámetros en la firma del método. Si los argumentos no están en la posición correcta y los tipos no coinciden, se generará un error de compilación.

En Visual C #, podemos especificar los parámetros por su nombre y, por lo tanto, proporcionar los argumentos en una secuencia que difiera del definido en la firma del método. Para utilizar argumentos nombrados, debemos proporcionar el nombre del parámetro y el valor correspondiente separados por dos puntos.

El siguiente ejemplo muestra como invocar el método **ShowMessage** utilizando argumentos nombrados para pasar el parámetro **iconID**.

```
ShowMessage("hello, World!", iconID:19);
```

Al utilizar argumentos nombrados en conjunto con parámetros opcionales, podemos omitir fácilmente los parámetros. Cualquier parámetro opcional recibirá su valor predeterminado. Sin embargo, si omitimos los parámetros obligatorios, el código no compilará.

Podemos mezclar los argumentos por posición y los argumentos nombrados. Sin embargo, debemos especificar todos los argumentos posicionales antes de los argumentos nombrados.

El siguiente ejemplo, muestra una forma incorrecta de invocar al método **ShowMessage** que hará que el código no compile debido a que los argumentos por posición deben ser especificados antes que los argumentos nombrados.

```
ShowMessage(iconID:19, "Hello!");
```

Para obtener más información acerca del uso de argumentos nombrados, se recomienda visitar el siguiente enlace:

Named and Optional Arguments (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/dd264739.aspx

Creando métodos que utilicen parámetros de salida out

Mediante la sentencia **return**, un método puede devolver un valor hacia el código que lo invoca. Si necesitamos devolver más de un solo valor al código que realiza la llamada, podemos utilizar los parámetros de salida para devolver los datos adicionales requeridos. Cuando agregamos un parámetro de salida a un método, en el cuerpo del método debemos asignar un valor a ese parámetro o de lo contrario obtendremos un error de compilación. Cuando el método finaliza, el valor del parámetro de salida es asignado a la variable que es especificada como el argumento correspondiente en la llamada al método.

Para definir un parámetro de salida, en la firma del método, anteponemos la palabra reservada **out** al tipo del parámetro. Un método puede tener tantos parámetros de salida como sean requeridos.

El siguiente ejemplo, muestra cómo definir un método que utiliza un parámetro de salida.

```
bool IsStatusError(out string statusMessage)
{
    bool FoundError = false;
    statusMessage = "";

    // Código para asignar valores de retorno FoundError y statusMessage
    // ...

    return FoundError;
}
```

Para utilizar un parámetro de salida, debemos proporcionar una variable para el argumento correspondiente en el momento de invocar al método. No es necesario asignarle un valor previamente debido a que el método invocado lo hará. También debemos anteponer la palabra clave **out** al argumento en la llamada. Si intentamos especificar un argumento que no sea una variable o si omitimos la palabra clave **out**, el código no compilará.

El siguiente ejemplo, muestra cómo invocar un método que acepta un parámetro de salida.

```
string StatusMessage;
bool IsError = IsStatusError(out StatusMessage);
```

Para obtener más información acerca de los parámetros de salida, se recomienda visitar el siguiente enlace:

out parameter modifier (C# Reference)
https://msdn.microsoft.com/en-us/library/ee332485.aspx

Lección 3:
Manejo de Excepciones

El manejo de excepciones es una estrategia importante para asegurar una buena experiencia de usuario y para limitar la perdida de datos. Las aplicaciones deben ser diseñadas con el manejo de excepciones en mente.

En esta lección, aprenderemos a implementar un manejo de excepciones efectivo en nuestras aplicaciones. Aprenderemos también, la forma de utilizar excepciones en nuestros métodos para indicar de forma elegante una condición de error al código que invoca nuestros métodos.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Describir el propósito de una excepción.
- Manejar excepciones utilizando un bloque **try/catch**.
- Utilizar un bloque **finally** para ejecutar código después de una excepción.
- Lanzar una excepción.

¿Qué es una Excepción?

Hay muchas cosas que pueden salir mal mientras se ejecuta una aplicación. Algunos errores pueden ocurrir debido a fallas en la lógica de la aplicación, pero otros pueden deberse a condiciones fuera del control de nuestra aplicación. Por ejemplo, nuestra aplicación no puede garantizar que exista un determinado archivo en el sistema de archivos o que una base de datos requerida se encuentre disponible. Una indicación de un error o de una condición que rara vez ocurre, es conocida como **Excepción**.

Cuando diseñemos una aplicación, debemos considerar la manera de garantizar que nuestra aplicación se pueda recuperar de la mejor forma cuando una excepción ocurra. Es una práctica común, comprobar simplemente el valor devuelto por el método para garantizar que se ha ejecutado correctamente, sin embargo, esta estrategia no siempre es suficiente para manejar todos los errores que pueden ocurrir debido a que:

- No todos los métodos devuelven un valor.
- En la mayoría de los casos, es necesario conocer por qué la llamada al método ha fallado y no sólo el hecho de que ha fallado.
- Los errores inesperados como, por ejemplo, quedarse sin memoria, no pueden ser manejados de esta manera.

Tradicionalmente, las aplicaciones utilizan la estrategia de un objeto de error global. En esta estrategia de manejo de errores, cuando una parte del código causa un error, los datos del error se almacenan en el objeto global para indicar la causa del mismo y luego se retorna al código que invocó al método que causó el error. Es responsabilidad del código que invoca, examinar el objeto de error y determinar cómo manejar la situación. Sin embargo, esta estrategia no es robusta debido a que es muy fácil para un programador olvidar el manejo de errores de forma apropiada.

¿Cómo se propagan las excepciones?

Un método puede lanzar una excepción cuando detecta que algo inesperado ha sucedido, por ejemplo, cuando la aplicación intenta abrir un archivo que no existe.

Cuando en un método se genera una excepción, el código debe estar preparado para detectar y manejar esta excepción. Si el código no procesa la excepción, la ejecución es interrumpida y la excepción se propaga automáticamente hacia el código que invocó al método que falló. Este proceso continúa hasta que una sección de código toma la responsabilidad para el manejo de la excepción. La ejecución continúa en esa sección de código después de que la lógica de manejo de excepciones se haya completado. Si ningún código maneja la excepción, entonces el proceso finalizará y se mostrará un mensaje al usuario.

El tipo Exception

Cuando una excepción ocurre, es conveniente incluir información acerca de la causa original para que el método que maneja la excepción pueda tomar la acción correctiva apropiada. En el .NET Framework, las excepciones se basan en la clase **Exception** la cual contiene información acerca de la excepción. Cuando un método dispara una excepción, se crea un objeto **Exception** y se puede proporcionar a través de él, información acerca de la causa del error. El objeto **Exception** es entonces pasado al código que maneja la excepción.

La siguiente tabla describe algunas de las clases **Exception** proporcionadas por el .NET Framework.

Clase Exception	Espacio de nombres	Descripción
Exception	System	Representa cualquier excepción que es disparada durante la ejecución de una aplicación. Esta clase sirve de base para crear nuestras propias clases **exception** personalizadas.
SystemException	System	Representa todas las excepciones disparadas por el CLR. La clase **SystemException** es la clase base de todas las clases **exception** en el espacio de nombres **System.**
ApplicationException	System	Representa todas las excepciones no fatales disparadas por las aplicaciones y no por el CLR.
NullReferenceException	System	Representa una excepción causada al intentar utilizar un objeto que es nulo.
FileNotFoundException	System.IO	Representa una excepción causada cuando un archivo no existe.
SerializationException	System.Runtime.Serialization	Representa una excepción causada durante el proceso de serialización o deserialización.

Para obtener más información acerca de la clase **Exception**, se recomienda visitar el siguiente enlace:

Exception Class
https://msdn.microsoft.com/en-us/library/system.exception.aspx

Manejando Excepciones utilizando el bloque Try/Catch

El bloque **try/catch** es la sentencia de programación clave que nos permite implementar el manejo estructurado de excepciones (Structured Exception Handling - SEH) en nuestras aplicaciones. Podemos encerrar el código que podría fallar y provocar una excepción dentro del bloque **try** y agregar uno o más bloques **catch** para controlar las excepciones que se puedan producir.

El siguiente ejemplo, muestra la sintaxis para definir un bloque **try/catch**.

```
try
{
    // Código que puede generar una excepción
}
catch([Tipo de excepción a manejar])
{
    // Código a ejecutar cuando ocurra un tipo de excepción
}
catch ([Tipo de excepción a manejar])
{
    // Código a ejecutar cuando ocurra otro tipo de excepción
}
```

Las instrucciones que se incluyen entre las llaves del bloque **try** pueden invocar métodos de otros objetos. Si cualquiera de las instrucciones dentro del bloque **try** provoca que una excepción sea lanzada, la ejecución pasa al bloque **catch** correspondiente. Podemos especificar bloques **catch** para diferentes tipos de excepciones. La especificación **catch** para cada bloque, determina la excepción que atrapará y, opcionalmente, la variable donde se almacenará la instancia de la excepción. Es una buena práctica incluir un bloque **catch** para el tipo de excepción general al final de los bloques **catch** para capturar todas las excepciones que no se hayan capturado en algunos de los bloques **catch** previos.

En el siguiente ejemplo, si el código en el bloque **try** causa una excepción **NullReferenceException**, el código en el bloque **catch** correspondiente será ejecutado. Si cualquier otro tipo de excepción ocurre, el código en el bloque **catch** para el tipo **Exception** será ejecutado.

```
try
{
    // Código que puede generar una excepción
}
catch(NullReferenceException ex)
{
    // Atrapa todas las excepciones NullReferenceException
}
catch (Exception ex)
{
    // Atrapa todas las demás excepciones
}
```

Cuando definamos más de un bloque **catch**, debemos asegurarnos de colocarlos en el orden correcto. Cuando se produce una excepción, el CLR intenta encontrar el bloque **catch** que atrapa la excepción que coincida con la excepción que haya sido generada. Debemos poner bloques **catch** más específicos antes de bloques **catch** menos específicos, de lo contrario el código no compilará.

Para obtener más información acerca del bloque **try/catch**, se recomienda visitar el siguiente enlace:

try-catch (C# Reference)
https://msdn.microsoft.com/en-us/library/0yd65esw(v=vs.140).aspx

Utilizando un bloque Finally

Algunos métodos pueden contener código crítico que siempre deba ser ejecutado, incluso si se produce una excepción. Por ejemplo, un método puede necesitar asegurarse de que un archivo que estaba escribiendo sea cerrado o bien liberar los recursos utilizados antes de terminar. Un bloque **finally** permite manejar esta situación.

Especificamos el bloque **finally** después de todos los bloques **catch**. El bloque **finally** especifica el código que debe ejecutarse cuando el bloque **try/catch** finalice independientemente de que se haya generado o no alguna excepción. Si una excepción es atrapada por un bloque **catch**, el código en el bloque **catch** será ejecutado antes que el bloque **finally**.

También podemos agregar un bloque **finally** a un bloque **try** que no tenga bloques **catch**. En este caso, las excepciones no son manejadas, pero el bloque **finally** siempre será ejecutado.

El siguiente ejemplo, muestra cómo implementar un bloque **try/catch/finally**.

```
try
{
    // Código que puede generar una excepción
}
catch(NullReferenceException ex)
{
    // Atrapa todas las excepciones NullReferenceException
}
catch (Exception ex)
{
    // Atrapa todas las demás excepciones
}
finally
{
    // El código en este bloque siempre será ejecutado,
    // haya o no haya excepciones.
}
```

Para obtener más información acerca del bloque **try/catch/finally**, se recomienda visitar el siguiente enlace:

try-catch-finally (C# Reference)
https://msdn.microsoft.com/en-us/library/dszsf989(v=vs.140).aspx

Lanzando Excepciones

Podemos crear una instancia de una clase **Exception** en nuestro código y lanzar la excepción para indicar que ha ocurrido una excepción. Cuando lanzamos una excepción, la ejecución del bloque de código actual termina y el CLR pasa el control al primer controlador de excepciones disponibles que atrape la excepción.

Para lanzar una excepción, utilizamos la palabra clave **throw** y especificamos el objeto excepción a lanzar.

El siguiente ejemplo, muestra cómo crear una instancia de la clase **NullReferenceException** para posteriormente lanzar el objeto **NullException**.

```
var NullException =
    new NullReferenceException("El valor proporcionado es nulo.");
throw NullException;
```

Una estrategia común es atrapar las excepciones en bloques **catch** y tratar de manejarlas. Si el bloque **catch** de una excepción no puede resolver el error, este puede volver a lanzar la excepción para propagarla al nivel superior del código que llamó al actual.

El siguiente ejemplo, muestra como volver a lanzar una excepción que ha sido atrapada en un bloque **catch**.

```
try
{
    // Código que puede generar una excepción
}
catch(NullReferenceException ex)
{
    // Atrapa todas las excepciones NullReferenceException
}
catch (Exception ex)
{
    // Trata de manejar la excepción.
    // ....
    // Si no se puede manejar la excepción, lanzarla al código
    // que invocó al método que contiene este código.
    throw;
}
```

Lección 4:
Monitoreo de aplicaciones

Cuando desarrollamos aplicaciones del mundo real, escribir código es simplemente una parte del proceso. Por lo regular, debemos dedicar una cantidad considerable de tiempo para resolver bugs, solucionar problemas y optimizar el rendimiento de nuestro código. Visual Studio y el .NET Framework, proporcionan varias herramientas que nos pueden ayudar a realizar estas tareas de forma más eficiente.

En esta lección, aprenderemos a utilizar diversas herramientas y técnicas para monitorear y resolver problemas en nuestras aplicaciones.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Utilizar las características de Registro (**logging**) y Rastreo (**tracing**) en el código de una aplicación.
- Utilizar la característica **Profiling** en Visual Studio.
- Utilizar los contadores de rendimiento para monitorear el rendimiento de una aplicación.

Utilizando Registro (Logging) y Seguimiento (Tracing)

Registro y seguimiento son dos conceptos similares pero distintos. Cuando implementamos logging en nuestra aplicación, agregamos código para escribir información hacia un medio de almacenamiento de registro, tal como un archivo de texto o el registro de eventos de Windows.

Logging nos permite proporcionar más información a los usuarios y administradores sobre lo que está haciendo nuestro código. Por ejemplo, si la aplicación atrapa una excepción, podemos escribir los detalles de la excepción en el registro de eventos de Windows para que el usuario o el administrador del sistema puedan resolver cualquier problema relacionado.

Por otro lado, los desarrolladores utilizan **Tracing** para monitorear la ejecución de una aplicación. Cuando implementamos **tracing**, agregamos código para escribir mensajes hacia un agente **Trace** que se encuentre escuchando mensajes, el agente a su vez dirige la salida a un destino especificado. En Visual Studio, de manera predeterminada, los mensajes de seguimiento aparecen en la ventana de salida (**Output**). Típicamente utilizamos **tracing** para proporcionar información acerca de los valores de las variables o el resultado de algunas expresiones para ayudarnos a determinar por qué nuestra aplicación se comporta de una manera particular.

También podemos utilizar técnicas **tracing** para interrumpir la ejecución de una aplicación en respuesta a las condiciones que nosotros definamos.

Escribiendo al Log de Eventos de Windows

Escribir al log de eventos de Windows es uno de los requerimientos más comunes de registro (***logging)*** que podríamos encontrar. La clase **System.Diagnostics.EvenLog** proporciona varios métodos estáticos que se pueden utilizar para escribir en log de eventos de Windows. En particular, el método **EventLog.WriteEntry** incluye varias sobrecargas que podemos utilizar para registrar varias combinaciones de información. Para escribir al log de eventos de Windows, necesitamos proporcionar al menos 3 piezas de información:

- **El log de eventos**. Este es el nombre del log de eventos de Windows hacia el cual queremos escribir. En la mayoría de los casos escribimos al log **Application**, sin embargo, podemos crear logs personalizados.

- **El origen del evento**. Esto identifica el origen del evento y típicamente suele ser el nombre de la aplicación. Cuando creamos un origen de eventos, este se asocia con un log de eventos. Para crear un origen de evento es necesario contar con permisos de administrador, por lo que es recomendable que el origen de evento y el log personalizado sean creados durante la instalación de la aplicación ya que durante su ejecución es probable que el usuario que ejecuta la aplicación no tenga privilegios de administrador. Si la aplicación no se ejecuta con suficientes permisos, una excepción **SecurityException** será disparada cuando la aplicación intente crear un origen de evento o trate de escribir al log de eventos.

- **El Mensaje**. Ese es el texto que deseamos agregar al registro.

También podemos utilizar el método **WriteEntry** para especificar una categoría, un ID de evento y de ser necesario, una severidad de evento.

El siguiente ejemplo, muestra cómo escribir un mensaje al log de eventos **Application** de Windows.

```
string EventLogName = "Application";
string EventSource = "Demo logging";
string EventMessage = "Hola mundo desde la aplicación Demo";
if (!EventLog.SourceExists(EventSource))
{
    EventLog.CreateEventSource(EventSource, EventLogName);
}
EventLog.WriteEntry(EventSource, EventMessage);
```

Para obtener más información acerca de cómo escribir al log de eventos de Windows, se recomienda visitar el siguiente enlace:

How to write to an event log by using Visual C#
https://support.microsoft.com/en-us/kb/307024

Depuración y Seguimiento (Debugging y Tracing)

El espacio de nombres **System.Diagnostic** incluye dos clases, **Debug** y **Trace**. Estas clases nos proporcionan información acerca del rendimiento de una aplicación durante su desarrollo o después de su implementación a producción. Estas clases son sólo una parte de las características de instrumentación que están disponibles en el.NET Framework.

Las clases **Debug** y **Trace** funcionan de forma similar e incluyen diversos métodos en común. Sin embargo, las instrucciones **Debug** sólo están activas si se compila la solución en modo de depuración, mientras que las instrucciones **Trace** están activas en ambos modos de compilación, **Debug** y **Realease**.

Las clases **Debug** y **Trace** incluyen métodos para escribir cadenas formateadas a la ventana **Output** en Visual Studio o hacia algún otro agente de escucha que hayamos configurado. También podemos escribir a la ventana **Output** solo cuando se cumpla cierta condición y podemos ajustar la identación de los mensajes **trace**. Por ejemplo, si estamos escribiendo hacia la ventana **Output** el detalle de cada objeto dentro de una enumeración, podríamos querer identar ese detalle para distinguirlo de otros mensajes de salida.

Las clases **Debug** y **Trace** también incluyen un método llamado **Assert**. El método **Assert** nos permite especificar una condición (una expresión que pueda ser evaluada como falsa o verdadera) junto con una cadena de formato. Si la condición se evalúa a **false**, el método **Assert** interrumpe la ejecución del programa y muestra una caja de diálogo modal con el mensaje que especifiquemos. Este método es útil si necesitamos identificar el punto en un programa de larga duración en el cual se ha presentado una condición inesperada.

El siguiente ejemplo muestra cómo utilizar la clase **Debug** para escribir mensajes a la ventana **Output** e interrumpir la ejecución si una condición inesperada se presenta.

```
int Option;
Console.Write("Por favor escribe una opción entre 1 y 10 y presiona <enter>: ");
string UserOption = Console.ReadLine();
Debug.Assert(int.TryParse(UserOption, out Option),
   string.Format($"imposible convertir '{UserOption}' como entero"));
Debug.WriteLine($"El valor proporcionado es {UserOption}");
Debug.WriteLine($"El valor de la opción es {Option}");
Console.WriteLine("Presiona <enter> para finalizar");
Console.ReadLine();
```

Para obtener más información acerca del uso de las clases **Trace** y **Debug**, se recomienda visitar el siguiente enlace:

How to trace and debug in Visual C#
https://support.microsoft.com/en-us/kb/815788

Perfilamiento (Profiling)

Cuando desarrollamos aplicaciones, hacer que nuestro código funcione sin errores es solo parte del desafío. También debemos garantizar que el código funcione eficientemente. Para ello es necesario revisar el tiempo que toma el código para realizar tareas y ver si utiliza recursos excesivos de procesador, memoria, disco o red.

Visual Studio incluye una gama de herramientas, conocidas colectivamente como ***Visual Studio Profiling Tools***, que nos pueden ayudar a analizar el rendimiento de nuestras aplicaciones. De manera general, ejecutar un análisis de rendimiento en Visual Studio consta de tres pasos:

1. **Crear y ejecutar una sesión de rendimiento**. Todos los análisis de rendimiento se llevan a cabo dentro de una sesión de rendimiento. Se puede crear y ejecutar una sesión de rendimiento lanzando el *Asistente de Rendimiento* desde el menú **Debug->Profiler->Performance Profiler…** de Visual Studio. Cuando la sesión de rendimiento está en ejecución, ejecutamos la aplicación como lo hacemos normalmente. Mientras se está ejecutando la aplicación, típicamente utilizamos la funcionalidad de nuestra aplicación que sospechamos pueda estar causando problemas de rendimiento.

2. **Analizar el informe de perfilamiento**. Cuando terminamos de ejecutar la aplicación, Visual Studio muestra el informe de perfilamiento. Esto incluye información que puede proporcionar ideas para mejorar el rendimiento de nuestra aplicación. Por ejemplo, podemos:
    a. Ver que funcionalidad consume más tiempo de CPU.
    b. Ver la línea de tiempo que muestra lo que la aplicación estaba haciendo.
    c. Ver las advertencias y sugerencias de cómo mejorar el código.

3. **Revisar el código y repetir el análisis**. Cuando el análisis haya concluido, debemos realizar los cambios en el código para arreglar cualquier problema que hayamos identificado. Después podemos ejecutar una nueva sesión de rendimiento y generar un nuevo informe de generación de perfilamiento. Las herramientas de **Visual Studio Profiling**, nos permiten comparar dos informes para ayudarnos a identificar y cuantificar cómo ha cambiado el rendimiento de nuestro código.

Las sesiones de rendimiento trabajan por muestreo. Cuando creamos una sesión de rendimiento, podemos elegir si deseamos un muestreo del uso de CPU, asignación de memoria, información de concurrencia en aplicaciones de múltiples hilos o si deseamos utilizar instrumentación para recopilar información detallada sobre cada llamada a funciones durante un lapso de tiempo. En la mayoría de los casos, se recomienda empezar utilizando el muestreo de CPU que es la opción predeterminada. El Muestreo de CPU utiliza sondeo estadístico para determinar qué funciones utilizan más tiempo de

CPU. Esto proporciona una visión del rendimiento de la aplicación, sin consumir muchos recursos y sin ralentizar nuestra aplicación.

Para obtener más información acerca de la característica de Perfilamiento en Visual Studio, se recomienda visitar el siguiente enlace:

Analyzing Application Performance by Using Profiling Tools
https://msdn.microsoft.com/library/z9z62c29.aspx

Contadores de rendimiento

Los contadores de rendimiento son herramientas del sistema que recopilan información sobre como son utilizados los recursos. Examinar los contadores de rendimiento puede proporcionarnos información adicional sobre lo que la aplicación está haciendo y puede ayudarnos a resolver problemas de rendimiento. Los contadores de rendimiento están organizados en 3 principales grupos:

- **Contadores que son proporcionados por el sistema operativo y la plataforma de hardware utilizada**. Este grupo incluye a los contadores que se pueden utilizar para medir el uso del procesador, uso de memoria física, uso de disco, y el uso de la red. El detalle de los contadores disponibles variará según el hardware del equipo.

- **Contadores que son proporcionados por el .Net Framework.** El .Net Framework incluye contadores que se pueden utilizar para medir una amplia gama de características de una aplicación. Por ejemplo, se puede ver el número de excepciones disparadas, ver el detalle bloqueos, el uso de threads y examinar el comportamiento del recolector de basura (garbage collector).

- **Contadores personalizados creados por nosotros mismos**. Podemos crear nuestros propios contadores de rendimiento para examinar aspectos específicos del comportamiento de la aplicación. Por ejemplo, podemos crear un contador de rendimiento para contar el número de llamadas a un método en particular o para contar el número de veces que una excepción específica es lanzada.

Navegando y utilizando contadores de rendimiento

Los contadores de rendimiento están organizados en categorías. Esto nos ayuda a encontrar los contadores que deseamos cuando capturamos y examinamos los datos de rendimiento. Por ejemplo, la categoría **PhysicalDisk** incluye normalmente contadores para el porcentaje de tiempo dedicado a leer y escribir a disco, la cantidad de datos leídos y escritos a disco, y la longitud de la cola para leer y escribir datos al disco.

Normalmente, capturamos y vemos los datos de los contadores de rendimiento en el Monitor de rendimiento (*perform.exe*). El monitor de rendimiento se incluye en el sistema operativo Windows y permite ver o capturar los datos de los contadores de rendimiento en tiempo real. Cuando utilizamos el Monitor de rendimiento, podemos navegar por las categorías de contadores y agregar varios contadores de rendimiento a una presentación gráfica. También podemos crear conjuntos de colectores de datos para capturar datos para reportes o análisis.

Es posible también navegar por los contadores de rendimiento disponibles en el equipo desde Visual Studio. En el Explorador de Servidores, expandimos los Servidores, expandimos el nombre del equipo y después expandimos Contadores de Rendimiento.

Crear contadores de rendimiento personalizados

Podemos utilizar las clases **PerformanceCounter** y **PerformanceCounterCategory** para interactuar con los contadores de rendimiento en diversas formas. Por ejemplo, podemos:

- Iterar sobre las categorías de contador de rendimiento disponibles en un equipo especificado.
- Iterar sobre los contadores de rendimiento dentro de una categoría especificada.
- Comprobar si existen categorías de contadores de rendimiento o contadores de rendimiento en la computadora local.
- Crear categorías de contadores de rendimiento personalizado o contadores de rendimiento.

Por lo general, debemos crear categorías de contadores de rendimiento personalizados y contadores de rendimiento durante el proceso de instalación en lugar de hacerlo durante la ejecución de la aplicación. Los contadores de rendimiento no deberían ser utilizados inmediatamente después de haberlos creado ya que existe un tiempo entre su creación y su habitación para ser utilizados.

Después de crear contadores de rendimiento personalizados en un equipo específico, estos permanecen ahí. No es necesario tener que volver a crearlos cada vez que se ejecute la aplicación. Para crear un contador de rendimiento personalizado, debemos especificar un tipo de contador base mediante la enumeración **PerformanceCounterType**.

El siguiente ejemplo, muestra cómo crear programáticamente una categoría de contadores de rendimiento personalizada. Este ejemplo crea una nueva categoría de contadores de rendimiento llamada ***NWTraders Ordenes***. La categoría contiene dos contadores de rendimiento. El primer contador registra el total de pedidos aprobados, y el segundo contador registra el total de pedidos rechazados.

```
if (PerformanceCounterCategory.Exists("NWTraders Ordenes"))
{
    Console.WriteLine("La categoría NWTraders Ordenes ya existe");
}
else
{
    var Contadores = new CounterCreationDataCollection();
    var Aprobados = new CounterCreationData();
    Aprobados.CounterHelp = "Total de pedidos aprobados";
    Aprobados.CounterName = "# Aprobados";

    Aprobados.CounterType = PerformanceCounterType.NumberOfItems32;
    Contadores.Add(Aprobados);
    var Rechazados = new CounterCreationData();
    Rechazados.CounterHelp = "Total de pedidos rechazados";
    Rechazados.CounterName = "# Rechazados";
    Rechazados.CounterType = PerformanceCounterType.NumberOfItems32;
    Contadores.Add(Rechazados);

    PerformanceCounterCategory.Create("NWTraders Ordenes",
```

```
                "Categoría demostrativa",
                PerformanceCounterCategoryType.SingleInstance,
                Contadores);
        }
```

Además de poder crear categorías y contadores de rendimiento programáticamente, también es posible hacerlo desde el Explorador de Servidores en Visual Studio.

Utilizando Contadores de Rendimiento Personalizados

Al crear contadores de rendimiento personalizados, nuestras aplicaciones deben proporcionar datos a los contadores de rendimiento. Los contadores de rendimiento proporcionan varios métodos que nos permiten actualizar su valor, tales como los métodos de incremento y decremento. La manera en que el contador procesará el valor dependerá del tipo base seleccionado al crear el contador.

El siguiente ejemplo, muestra como modificar contadores de rendimiento de forma programática.

```
// Obtenemos la referencia a los contadores
var Aprobados = new PerformanceCounter(
    "NWTraders Ordenes", "# Aprobados", false);
var Rechazados = new PerformanceCounter(
    "NWTraders Ordenes", "# Rechazados", false);
// Asignar un valor
Aprobados.RawValue = 20;
// Incrementar su valor
Rechazados.Increment();
// Imprimimos los valores
Console.WriteLine(
    $"Aprobados {Aprobados.NextValue()}, Rechazados {Rechazados.NextValue()}");
```

Una vez que hayamos creado una categoría de contador de rendimiento personalizado, podemos ir a la categoría y seleccionar los contadores de rendimiento individuales en el Monitor de rendimiento. Cuando ejecutemos la aplicación, podemos utilizar el Monitor de rendimiento para ver los datos de los contadores de rendimiento personalizados en tiempo real.

Introducción a C#

Módulo 3: Desarrollando el código para una aplicación gráfica

Acerca del módulo

Para crear aplicaciones gráficas efectivas utilizando Windows Presentation Foundation (WPF) u otras tecnologías .NET, primero debemos aprender algunas construcciones básicas de Visual C#. Por ejemplo, necesitamos conocer cómo crear estructuras simples para representar los elementos de datos con los que estamos trabajando. Necesitamos saber cómo organizar esas estructuras dentro de colecciones para que podamos agregar, eliminar, recuperar o iterar sobre esos elementos. Finalmente, necesitamos conocer la forma de suscribirnos a eventos para que podamos responder a las acciones de los usuarios.

En este módulo, aprenderemos a crear y utilizar estructuras y enumeraciones, organizar datos en colecciones, y crear y suscribirse a eventos.

Objetivos

Al finalizar este módulo, los participantes contarán con las habilidades y conocimientos para:

- Crear y utilizar estructuras y enumeraciones.
- Utilizar clases de colecciones para organizar datos.
- Crear y suscribirse a eventos.

Los temas que se cubren en este módulo son:

- Lección 1: Implementando Estructuras y Enumeraciones.
- Lección 2: Organizando datos dentro de colecciones.
- Lección 3: Manejando Eventos.

Lección 1:
Implementando Estructuras y Enumeraciones

El .NET Framework incluye varios tipos de datos predefinidos, tales como **Int32**, **Decimal**, **String** y **Boolean** entre otros. Sin embargo, supongamos que queremos crear un objeto que represente a un Vehículo. ¿Qué tipo de dato utilizaríamos? Podríamos utilizar los tipos de datos predefinidos para representar las propiedades de un Vehículo, tal como su marca (un String) o su kilometraje (un Int32). Sin embargo, necesitamos una manera de representar a un Vehículo como una entidad, de tal forma que podamos realizar acciones tal como agregar un Vehículo a una colección o comparar un Vehículo con otro.

En esta lección, aprenderemos a utilizar estructuras y enumeraciones para crear nuestros propios tipos simples de datos.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Crear y utilizar enumeraciones.
- Crear y utilizar estructuras.
- Definir constructores para inicializar estructuras.
- Crear propiedades para obtener y establecer el valor de un campo en una estructura.
- Crear indexadores para exponer miembros de una estructura mediante el uso de índices enteros.

Creando y utilizando Enumeraciones

Un tipo enumeración o **enum**, es una estructura que nos permite crear una variable con un conjunto fijo de posibles valores. Un tipo enumeración consiste de un conjunto de constantes con nombre denominado lista de enumeradores.

El ejemplo más común es utilizar una enumeración para definir el día de la semana. Sólo hay siete posibles valores para los días de la semana y podemos estar seguros de que estos valores nunca cambiarán. Para definir esos valores, podemos utilizar un tipo enumeración que se declara mediante la palabra clave **enum**.

El siguiente ejemplo muestra cómo crear un tipo **enum**.

```
enum Dia
{
    Domingo, Lunes, Martes, Miercoles, Jueves, Viernes, Sabado
}
```

Para utilizar la enumeración, creamos una instancia de una variable del tipo **enum** creado y especificamos qué miembro de la enumeración deseamos utilizar.

El siguiente ejemplo muestra cómo utilizar un tipo **enum**.

```
Dia DiaFavorito = Dia.Miercoles;
```

Utilizar enumeraciones tiene varias ventajas sobre el uso de texto o tipos numéricos:

- **Administración mejorada**. Al restringir una variable a un conjunto fijo de valores válidos, hay menos riesgo de experimentar argumentos no válidos y a errores al escribir valores ya que se especifica claramente al código cliente qué valores son válidos para la variable.
- **Experiencia de desarrollador mejorada**. En Visual Studio, la característica **IntelliSense** nos indica los valores disponibles cuando se utiliza una enumeración.
- **Legibilidad del código mejorada**. La sintaxis en el uso de enumeraciones hace que nuestro código sea más fácil de leer y entender.

Cada miembro de una enumeración tiene un ***Nombre*** y un ***Valor***. El nombre es el texto que definimos en las llaves, tal como *Lunes* o *Martes*. De manera predeterminada, el valor es un entero. Si no especificamos un valor para cada miembro, les es asignado un valor incremental iniciando desde 0. De forma predeterminada, el primer enumerador tiene el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1. En nuestro ejemplo, **Dia.Domingo** tiene el valor 0 y **Dia.Lunes** tiene el valor 1.

El siguiente ejemplo, muestra cómo podemos utilizar nombres y valores intercambiablemente.

```
// Establecer el valor a una variable enum por Nombre
Dia DiaFavorito = Dia.Viernes;
```

```
// Establecer el valor a una variable enum por Valor
Dia OtroDiaFavorito = (Dia)7;
```

Tipos enum con marcadores de Bits

Es posible utilizar un tipo enumeración para definir marcadores de bits. El uso de los marcadores de bits permite que una instancia del tipo de enumeración almacene cualquier combinación de los valores definidos en la lista de enumeradores. En otras palabras, podemos hacer que una sola variable pueda almacenar más de un valor de la enumeración.

Para crear una enumeración de marcadores de bits, debemos aplicar el atributo **System.FlagsAttribute** a la definición de la enumeración. Los valores de la enumeración deben ser definidos de tal forma que se pueda realizar con ellos las operaciones bit a bit de tipo AND, OR, NOT y XOR.

El siguiente ejemplo, define una enumeración como marcadores de bits.

```
[Flags]
enum Pasatiempo
{
    Lectura = 1, // 0001
    Deporte = 2, // 0010
    Musica = 4,  // 0100
    Cine = 8,    // 1000
}
```

El siguiente ejemplo muestra como asignar los valores **Lectura** y **Musica** a una variable de tipo **PasaTiempo**. Podemos notar que para hacer esto, utilizamos el operador **|** (OR) de bits.

```
Pasatiempo Pasatiempos = Pasatiempo.Lectura | Pasatiempo.Musica;
```

Internamente la variable Pasatiempos tendría la combinación de bits **0101**. Podemos notar que el bit de Lectura está prendido y el bit de Musica también está prendido.

El siguiente ejemplo, muestra un valor de enumerador no adecuado para la enumeración ya que su combinación de bits representa los valores **Musica** y **Lectura**.

```
Danza = 5 // 0101
```

En una enumeración de marcadores de bits, se recomienda incluir una constante con nombre de valor cero que signifique que "no se ha establecido ningún marcador". No es recomendable asignar a un marcador el valor cero si éste no significa que "no se ha establecido ningún marcador".

El siguiente ejemplo, muestra un valor que indica que el usuario no tiene pasatiempos.

```
Ninguno = 0, // 0000
```

El siguiente ejemplo, verifica si una variable contiene un valor especifico mediante el uso del operador **&** (AND) de bits.

```
if ((Pasatiempos & Pasatiempo.Lectura) == Pasatiempo.Lectura)
{
    Console.WriteLine("Le gusta leer");
}
```

Utilizando métodos de la clase System.Enum para descubrir y manipular valores Enum

Todas las enumeraciones son instancias del tipo **System.Enum**. No podemos derivar clases nuevas de **System.Enum**, pero podemos utilizar sus métodos para detectar información relacionada y manipular los valores de una instancia de enumeración.

El siguiente ejemplo, muestra el uso de algunos de los métodos de la clase **System.Enum**.

```
// Convertir un Nombre de enumerador a su Valor
var DiaFavorito = (Dia)Enum.Parse(typeof(Dia), "Martes");

// Obtener el Nombre de un enumerador
var NombreDia3 = Enum.GetName(typeof(Dia), 3);

// Recorrer los Valores de una enumeración
foreach (int i in Enum.GetValues(typeof(Dia)))
{
    // ...
}

// Recorrer los Nombres de una enumeración
foreach (string s in Enum.GetNames(typeof(Dia)))
{
    // ...
}
```

Es importante mencionar que como ocurre con cualquier constante, todas las referencias a los valores individuales de una enumeración se convierten en literales numéricas en tiempo de compilación. Al modificar una enumeración, debemos volver a compilar el código fuente que utilice dicha enumeración.

Para obtener más información acerca de los tipos **enum**, se recomienda visitar el siguiente enlace:

Enumeration Types (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/cc138362(v=vs.140).aspx

Creando y utilizando Estructuras

En Visual C#, un **struct** es una construcción de programación que se puede utilizar para definir tipos personalizados. Los tipos **struct**, son esencialmente estructuras de datos ligeras que representan piezas de información relacionadas como un solo elemento. Algunos ejemplos de estructuras son:

- Una estructura llamada **Punto** podría constar de campos para representar una coordenada X y una coordenada Y en un plano cartesiano.
- Una estructura llamada **Circulo** podría constar de campos para representar una coordenada X, una coordenada Y y un Radio.
- Una estructura llamada **Color** podría constar de campos para representar un componente Rojo, un componente Verde, y un componente Azul.

La mayoría de los tipos integrados en Visual C#, tales como **int**, **bool** o **char**, están definidos por estructuras. Podemos utilizar estructuras para crear nuestros propios Tipos que se comporten como los tipos integrados.

Para crear una estructura utilizamos la palabra clave **struct** como se muestra en el siguiente ejemplo.

```
public struct Punto
{
    public int X;
    public int Y;
}
```

La palabra clave **struct** puede ser precedida por un modificador de acceso - **public** en nuestro ejemplo - que específica donde puede ser utilizado ese tipo. En la declaración de una estructura podemos utilizar los siguientes modificadores de acceso.

- **public**. El modificador **public** hace que el tipo esté disponible para el código de cualquier Assembly.
- **internal**. El modificador **internal** hace que el tipo esté disponible para cualquier código dentro del mismo Assembly pero que no esté disponible en el código de otro Assembly. Este es el valor predeterminado si no especificamos un modificador de acceso.
- **private**. El modificador **private** hace que el tipo solo esté disponible para el código dentro de la estructura de datos que lo contiene. Sólo se puede utilizar el modificador de acceso privado con estructuras de datos anidadas.

Las estructuras pueden contener diversos miembros, incluyendo Constructores, Constantes, Campos, Propiedades, Métodos, Indizadores, Operadores, Eventos y tipos anidados.

Para crear una instancia de un tipo **struct**, utilizamos la palabra clave **new** como se muestra en el siguiente ejemplo.

```
Punto p = new Punto();
p.X = 5;
p.Y = 10;
```

Los tipos **struct** comparten la misma sintaxis que las clases, aunque están más limitadas que estas, por ejemplo:

- Los tipos **struct** son tipos valor y las clases son tipo referencia.
- Los tipos **struct** se copian en la asignación. Cuando se asigna un **struct** a una nueva variable, se copian todos los datos, y cualquier modificación que se realice en la nueva copia no afecta a los datos de la copia original.
- Dentro de una declaración **struct**, los campos no se pueden declarar e inicializar en la misma sentencia a menos que se declaren como constantes o estáticos. La siguiente declaración hará que el código no compile.

```
public struct Punto
{
        public int X=0;
        public int Y=0;
}
```

- Un tipo **struct** no puede declarar un constructor predeterminado (es decir, un constructor sin parámetros) ni un destructor. Un tipo **struct** solo puede declarar constructores que tengan parámetros.
- A diferencia de las clases, es posible crear instancias de un tipo **struct** sin utilizar el operador **new**. Las siguientes líneas compilan sin problema.

```
Punto p;
p.X = 5;
p.Y = 10;
```

- Un tipo **struct** puede implementar una interfaz, pero no puede heredar de otro **struct** o clase, ni puede ser la base de una clase, por esa razón, los miembros de tipos struct no se pueden declarar como **protected**.
- Los tipos **struct** heredan directamente de **System.ValueType**, que a su vez hereda de **System.Object**.
- Un tipo **struct** se puede utilizar como tipo que acepta valores **null** y se le puede asignar un valor **null**.

```
Punto? p;
p = null;
```

Inicializando Estructuras

Podemos observar que la sintaxis para inicializar un tipo **struct**, por ejemplo, **new Punto()**, es similar a la sintaxis para invocar a un método. Esto es porque cuando se crea una instancia de un tipo **struct**, en realidad se está invocando a un tipo especial de método llamado *constructor*. Un constructor es un método en el tipo **struct** que tiene el mismo nombre que el tipo **struct**.

Cuando instanciamos una estructura sin argumentos, como **new Punto()**, estamos invocando al constructor predeterminado que es creado por el compilador de Visual C#. Si queremos ser capaces de especificar los valores predeterminados de los campos al crear una instancia de una estructura, podemos agregar a la estructura constructores que acepten parámetros.

El siguiente ejemplo muestra cómo crear un constructor en una estructura.

```
public struct Punto
{
    public Punto(int cx, int cy)
    {
        X = cx;
        Y = cy;
    }

    public int X;
    public int Y;
}
```

Podemos ahora crear instancias de la estructura **Punto** utilizando el constructor creado como se muestra en el siguiente ejemplo.

```
Punto C = new Punto(4, 5);
```

Podemos agregar múltiples constructores a una estructura, cada constructor aceptando una combinación diferente de parámetros. Sin embargo, no podemos agregar un constructor predeterminado a la estructura, ya que este es creado por el compilador.

En caso de agregar un constructor, este debe inicializar cada uno de los campos porque de lo contrario obtendremos un error de compilación.

El siguiente código generaría un error de compilación debido a que no estamos inicializando todos los campos.

```
public struct Circulo
{
    public Circulo(int cx, int cy)
    {
        X = cx;
        Y = cy;
    }
```

```
        public int X;
        public int Y;
        public int Radio;
    }
```

Para asegurar que todos los campos sean inicializados y evitar el mensaje de error del compilador, podemos invocar a algún constructor que inicialice todos los campos o de no existir dicho constructor, podemos invocar al constructor predeterminado como se muestra en el siguiente ejemplo.

```
    public struct Circulo
    {
        public Circulo(int cx, int cy):this()
        {
            X = cx;
            Y = cy;
        }

        public int X;
        public int Y;
        public int Radio;
    }
```

De esta forma, el código compilará sin error.

Creando Propiedades

Es una práctica recomendada que los campos de una estructura o clase sean privados y no públicos. Sin embargo, al hacerlos privados, debemos contar con un mecanismo que nos permita establecer y obtener sus valores.

En Visual C#, una *Propiedad* es una construcción de programación que permite al código cliente obtener o establecer el valor de los campos privados dentro de una estructura o una clase. Para los consumidores de nuestra estructura o clase, la propiedad se comporta como un campo público. Dentro de nuestra estructura o clase, la propiedad es implementada mediante el uso de *accesores* que son un tipo especial de método. Una propiedad puede incluir uno o ambos de los siguientes *accesores*:

- Un accesor **get** para proporcionar acceso de lectura a un campo.
- Un accesor **set** para proporcionar acceso de escritura a un campo.

El siguiente ejemplo muestra cómo implementar una propiedad en una estructura.

```
public struct Cuadrado
{
    private decimal Lado_BF;

    public decimal Lado
    {
        get { return Lado_BF; }
        set { Lado_BF = value; }
    }
}
```

Dentro de la propiedad, los accesores **get** y **set** utilizan la siguiente sintaxis:

- El accesor **get** utiliza la palabra clave **return** para devolver al invocador el valor del campo privado.
- El accesor **set** utiliza una variable local especial llamada **value** para establecer el valor del campo privado. La variable **value** contiene el valor proporcionado por el código cliente que accede a la propiedad.

El siguiente ejemplo, muestra cómo utilizar una propiedad.

```
Cuadrado C = new Cuadrado();

// El siguiente código invoca al accesor set.
C.Lado = 5;
// El siguiente código invoca al accesor get.
decimal Lado = C.Lado;
```

El código cliente utiliza la propiedad tal como si fuera un campo público. Sin embargo, el utilizar propiedades públicas para exponer los campos privados ofrece ventajas sobre el uso de campos públicos directamente:

- Podemos utilizar propiedades para controlar el acceso externo a nuestros campos. Una propiedad que sólo incluye un accesor **get** es de sólo lectura, mientras que una propiedad que incluye sólo un accesor **set** es de sólo escritura.

```
private decimal Base_BF;
private decimal Altura_BF;

// Esta es una propiedad de solo lectura.
public decimal Base
{
    get { return Base_BF; }
}

// Esta es una propiedad de solo escritura.
public decimal Altura
{
    set { Altura_BF = value; }
}
```

- Podemos cambiar la implementación de las propiedades sin afectar el código del cliente. Por ejemplo, podemos agregar lógica de validación, o invocar a un método en lugar de leer el valor de un campo.

```
private decimal Lado_BF;
public decimal Lado
{
    get { return Lado_BF; }
    set
    {
        if(value < 1)
        {
            Lado_BF = 1;
        }
        else if(value > 5)
        {
            Lado_BF = 5;
        }
        else
        {
            Lado_BF = value;
        }
    }
}
```

- Las propiedades son requeridas para enlace a datos en WPF. Por ejemplo, podemos enlazar controles a los valores de una propiedad, pero no podemos enlazar controles a los valores de campos.

Cuando deseamos crear una propiedad que sólo obtiene y establece el valor de un campo privado sin realizar ninguna lógica adicional, podemos utilizar una sintaxis abreviada gracias a una característica del compilador de Visual C# conocida como Propiedades automáticamente implementadas.

- Para crear una propiedad que lee y escribe en un campo privado, podemos especificar el accesor **get** y **set** con la siguiente sintaxis:

```
public struct Cuadrado
{
    public decimal Lado { get; set; }
}
```

- Para crear una propiedad que solo lea un campo privado, podemos definir únicamente el accesor **get** o agregar el modificador de acceso **private** o **protected** al accesor **set**.

```
public struct Rectangulo
{
    // Propiedad auto implementada de solo lectura.
    public decimal Base { get; }

    // Propiedad auto implementada de solo lectura con accesor set privado.
    public decimal Altura { get; private set; }
}
```

- Para crear una propiedad que solo escriba en un campo privado, podemos definir únicamente el accesor **set** y agregar el modificador de acceso **private** o **protected** al accesor **get**.

```
public class Cuadrado
{
    // propiedad auto implementada de solo escritura.
    public decimal Lado { private get; set; }
}
```

Cuando utilizamos propiedades automáticamente implementadas, el compilador creará implícitamente un campo privado y lo mapeará a nuestra propiedad. Nosotros podemos cambiar la implementación de la propiedad en cualquier momento.

Para obtener más información sobre la restricción de los accesores, se recomienda visitar el siguiente enlace:

Restricting Accessor Accessibility (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/75e8y5dd(v=vs.140).aspx

Creando Indizadores

En algunos escenarios, es posible que deseemos utilizar una estructura o una clase como un contenedor para un arreglo de valores. Por ejemplo, podríamos crear una estructura para representar las distintas categorías de productos de un almacén. La estructura podría utilizar un arreglo de cadenas para almacenar la lista de categorías.

El siguiente ejemplo muestra una estructura que incluye un arreglo.

```
public struct Almacen
{
    public string[] Categorias { get; set; }
    public Almacen(string[] categorias)
    {
        Categorias = categorias;
    }
}
```

Cuando exponemos el arreglo como una propiedad pública, podemos utilizar la siguiente sintaxis para obtener una categoría de la lista.

```
var Almacen = new Almacen(
    new string[] { "Bebidas", "Lácteos", "Vegetales", "Condimentos" });

string PrimeraCategoria = Almacen.Categorias[0];
```

Una manera más intuitiva sería poder acceder a la primera categoría del almacén utilizando la siguiente sintaxis:

```
string PrimeraCategoria = Almacen[0];
```

Podemos hacer esto mediante la creación de un indizador (también conocido como Indexador). Un indizador es similar a una propiedad, ya que utiliza accesores **get** y **set** para controlar el acceso a un campo. Más importante aún, un indizador permite acceder a los miembros de la colección directamente desde el nombre del contenedor de la instancia de la estructura o clase proporcionando un valor de índice. Al igual que en los métodos, el tipo del índice puede ser un valor entero, aunque también podría ser de cualquier otro tipo, por ejemplo, un tipo **string**. Un indizador también puede aceptar más de un parámetro como el caso de las matrices.

Para declarar un indizador, utilizamos la palabra clave **this**, esto indica que la propiedad será accedida mediante el nombre de la instancia de la estructura.

El siguiente ejemplo muestra la estructura **Almacen** definiendo ahora un indizador.

```
public struct Almacen
{
    // Categorias ahora es un campo privado
    private string[] Categorias;
    public Almacen(string[] categorias)
```

```
        {
            Categorias = categorias;
        }

        // Este es el Indizador
        public string this[int index]
        {
            get { return Categorias[index]; }
            set { Categorias[index] = value; }
        }

        // Permitimos al código cliente determinar el tamaño de la colección.
        public int Length
        {
            get { return Categorias.Length; }
        }
    }
```

Cuando utilizamos un indizador para exponer un arreglo, utilizamos la siguiente sintaxis para obtener una categoría de la lista.

```
string PrimeraCategoria = Almacen[0];
int NumeroDeCategorias = Almacen.Length;
```

Al igual que una propiedad, podemos personalizar los accesores **get** y **set** en un indizador sin afectar el código del cliente. Podemos crear un indizador de sólo lectura haciendo disponible sólo un accesor **get**, y podemos crear un indizador de sólo escritura haciendo disponible sólo un accesor **set**.

Para obtener más información acerca de los indizadores, se recomienda visitar el siguiente enlace:

Using Indexers (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/2549tw02(v=vs.140).aspx

Lección 2:
Organizando datos dentro de colecciones

Cuando creamos múltiples elementos del mismo tipo, independientemente de que sean enteros, cadenas o tipos personalizados como un tipo **Persona**, necesitamos una forma de manejar estos elementos en conjunto. Necesitamos poder contar el número de elementos del conjunto, agregar elementos o eliminar elementos del conjunto, e iterar a través del conjunto un elemento a la vez. Para hacer esto, podemos utilizar una colección.

Las colecciones son una herramienta esencial para administrar múltiples elementos. También son muy importantes al desarrollar aplicaciones gráficas. Controles tales como cajas de lista desplegables y menús son típicamente colecciones enlazadas a datos.

En esta lección, aprenderemos a utilizar una variedad de clases colección en Visual C#.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Seleccionar colecciones apropiadas para diferentes escenarios.
- Manejar elementos en una colección.
- Utilizar la sintaxis Language Integrated Query (LINQ) para realizar consultas sobre una colección.

Seleccionando Colecciones

Todas las clases de Colección comparten varias características en común. Para administrar una colección de elementos, debemos ser capaces de:

- Agregar elementos a la colección.
- Eliminar elementos de la colección.
- Recuperar elementos específicos de la colección.
- Contar el número de elementos de la colección.
- Iterar a través de los elementos de la colección, un elemento a la vez.

Cada clase colección en Visual C# proporciona métodos y propiedades que soportan estas operaciones fundamentales. Sin embargo, más allá de estas operaciones, desearemos utilizar colecciones de diferentes formas dependiendo de las necesidades específicas de nuestra aplicación. Las clases **Collection** en Visual C# se dividen en varias categorías:

- **List**. Las clases Lista (**List)** almacenan colecciones lineales de elementos. Podemos pensar de una clase List como un arreglo de una dimensión que se expande dinámicamente a medida que se agregan elementos. Por ejemplo, podríamos utilizar una clase **List** para mantener una lista de Productos de un almacén.

- **Dictionary**. Las clases Diccionario (**Dictionary**), almacenan una colección de parejas Llave/valor. Cada elemento de la colección se compone de dos objetos - La llave y el valor. El valor es el objeto que deseamos almacenar y recuperar y la llave es el objeto que se utiliza para indexar y buscar el valor. En la mayoría de las clases diccionario, la llave debe ser única, mientras que los valores duplicados son perfectamente aceptables. Por ejemplo, podríamos utilizar una clase diccionario para mantener una lista de recetas de platillos. La llave podría contener el nombre único del platillo y el valor podría contener los ingredientes y las instrucciones para preparar el Platillo.

- **Queue**. Las clases Cola (**Queue**) representan una colección de objetos en una estructura FIFO, esto es, “primero en entrar, primero en salir – First input, first output”. Los elementos se recuperan de la colección en el mismo orden en que fueron agregados. Por ejemplo, se puede utilizar una clase Queue para procesar los pedidos de una tienda para asegurar que los clientes reciban sus pedidos en su turno apropiado.

- **Stack**. Las clases Pila (**Stack**) representan una colección de objetos en una estructura LIFO, esto es, “ultimo en entrar, primero en salir – Last input, first output”. El elemento que se agrega al último en la colección es el primer elemento que se recupera. Por ejemplo, podríamos utilizar una clase Stack para determinar los 10 visitantes más recientes de una tienda.

Cuando elijamos una clase colección del .NET Framework para un escenario específico, debemos hacernos las siguientes preguntas:

- ¿Necesitamos una Lista, un Diccionario, una Pila o una Cola?
- ¿Necesitaremos ordenar la colección?
- ¿Qué tan grande esperamos que crezca la colección?
- Si necesitamos una clase Diccionario, ¿Necesitamos recuperar los elementos tanto por índice como por la llave?
- ¿Nuestra colección consiste únicamente de cadenas?

Si podemos contestar todas estas preguntas, seremos capaces de seleccionar la clase colección de Visual C# que mejor se adapte a nuestras necesidades.

Clases Colección Estándares

El espacio de nombres **System.Collections** ofrece una amplia gama de colecciones de propósito general que incluye listas, diccionarios, colas y pilas. La siguiente tabla muestra las clases colección más importantes dentro del espacio de nombres **System.Collections**.

Clase	Descripción
ArrayList	**ArrayList** es una lista de propósito general que almacena una colección lineal de objetos. **ArrayList** incluye métodos y propiedades que permiten agregar elementos, remover elementos, contar el número de elementos de la colección y ordenar la colección.
BitArray	**BitArray** es una clase *Lista* que representa una colección de bits como valores booleanos. **BitArray** es comúnmente utilizada para operaciones de bits y aritmética booleana. Incluye métodos para realiza operaciones booleanas comunes como AND, OR, NOT y XOR.
Hashtable	La clase **Hashtable** es una clase *Diccionario* de propósito general que almacena una colección de pares Llave/Valor. **Hashtable** incluye métodos y propiedades que nos permiten recuperar elementos por llave, agregar elementos, quitar elementos y verificar la existencia de llaves y valores específicos dentro de la colección.
Queue	La clase **Queue** es una colección de objetos de tipo "*primero en entrar, primero en salir*". La clase **Queue** incluye métodos para agregar objetos a la cola (**Queue**) y recuperar objetos desde la cola (**Dequeue**).
SortedList	La clase **SortedList** almacena una colección de parejas Llave/Valor que están ordenadas por llave. Adicionalmente a la funcionalidad que proporciona la clase **Hashtable**, la clase **SortedList** permite recuperar los elementos ya sea por llave o por índice.
Stack	La clase **Stack** representa una colección de objetos "*último en entrar, primero en salir*". **Stack** incluye métodos para obtener el primer elemento de la colección sin eliminarlo (**Peek**), agregar un elemento en la parte superior de la pila (**Push**), además obtener y eliminar el elemento en la parte superior de la pila (**Pop**).

Para obtener más información acerca de las clases colección, se recomienda visitar el siguiente enlace:

System.Collections Namespace
https://msdn.microsoft.com/en-us/library/system.collections.aspx

Clases de Colecciones Especializadas

El espacio de nombres **System.Collections.Specialized** proporciona clases colección que son adecuadas para los requerimientos más especializados tales como las colecciones de diccionarios especializados y colecciones de cadenas fuertemente tipadas. La siguiente tabla muestra las clases colección más importantes en el espacio de nombres **System.Collections.Specialized**.

Clase	Descripción
ListDictionary	**ListDictionary** es una clase diccionario que está optimizada para colecciones pequeñas. Como regla general, si la colección incluye 10 elementos o menos, utilicemos un **ListDictionary**. Si la colección es más grande, utilicemos una **Hashtable.**
HybridDictionary	**HybridDictionary** es una clase diccionario que se puede utilizar cuando no se puede estimar el tamaño de la colección. **HybridDictionary** utiliza una implementación **ListDictionary** cuando el tamaño de la colección es pequeño de 10 o menos elementos y cambia a una implementación **Hastable** cuando la colección aumenta su tamaño.
NameValueCollection	**NameValueCollection** es una clase diccionario indexada en la cual la llave y el valor son cadenas. **NameValueCollection** generará un error si se intenta establecer una llave o un valor a cualquier tipo que no sea una cadena. Se puede recuperar elementos por llave o por índice.
OrderDictionary	**OrderDictionary** es una clase diccionario indexada que permite recuperar elementos por su llave o por su índice. A diferencia de la clase **SortedList**, los elementos de un **OrderedDictionary** no están ordenados por llave.
StringCollection	**StringCollection** es una clase de lista en la que cada elemento de la colección es una cadena. Podemos utilizar esta clase cuando deseemos almacenar una simple colección lineal de cadenas.
StringDictionary	**StringDictionary** es una clase diccionario en la que la llave y el valor son cadenas. A diferencia de la clase **NameValueCollection**, no podemos recuperar elementos en un **StringDictionary** por su índice.
BitVector32	**BitVector32** es un tipo Struct que puede representar un valor de 32 bits tanto como una colección de bits o como un valor entero. A diferencia de la clase **BitArray**, que puede expandirse indefinidamente, la estructura **BitVector32** es de un tamaño fijo de 32 bits. Como resultado, el **BitVector32** es más eficiente que el **BitArray** para valores pequeños. Se puede dividir una instancia **BitVector32** en secciones para almacenar de manera eficiente múltiples valores.

Para obtener más información acerca de las clases colección especializadas, se recomienda visitar el siguiente enlace:

System.Collections.Specialized Namespace
https://msdn.microsoft.com/en-us/library/system.collections.specialized(v=vs.110)

Utilizando Colecciones de tipo Lista

La colección de tipo lista más utilizada es la clase **ArrayList**. **ArrayList** almacena elementos como una colección lineal de objetos. Podemos agregar objetos de cualquier tipo en una colección **ArrayList**, **ArrayList** representa cada elemento en la colección como una instancia de **System.Object**. Cuando agregamos elementos a una colección **ArrayList**, **ArrayList** convierte implícitamente los elementos a un tipo **Object**. Al recuperar elementos de la colección, debemos convertir el objeto a su tipo original.

El siguiente ejemplo, muestra como agregar y recuperar elementos de una colección **ArrayList**.

```
// Crear una colección ArrayList.
ArrayList AL = new ArrayList();

// Agregar algunos elementos a la colección.
// Los elementos son implicitamente convertidos al tipo Object.
AL.Add(5);
AL.Add(DateTime.Now);
AL.Add("Hola, Mundo");

// Recuperar elementos desde la colección.
// Debemos convertir explicitamente al tipo original.
int i = (int)AL[0];
DateTime d = (DateTime)AL[1];
string s = AL[2].ToString();
```

Cuando trabajamos con colecciones, una de las tareas comunes de programación será iterar sobre la colección. Esencialmente, esto significa recuperar cada elemento de la colección, uno a la vez, para procesar su contenido. Para iterar sobre los elementos de la colección utilizamos el ciclo **foreach**. El ciclo **foreach** expone cada elemento de la colección, uno a la vez, utilizando el nombre de la variable especificada en la declaración del ciclo.

El siguiente ejemplo, muestra como iterar sobre una colección **ArrayList**.

```
foreach (object Elemento in AL)
{
    Console.WriteLine(Elemento);
}
```

El siguiente ejemplo, muestra otra colección **ArrayList** con elementos del mismo tipo. Podemos notar que el ciclo **foreach** realiza la conversión al tipo correcto. En este caso no se genera un error en tiempo de ejecución debido a que todos los elementos son del mismo tipo **Categoria**.

```
var Categorias = new ArrayList();

var Lacteos = new Categoria(1, "Lácteos");
var Condimentos = new Categoria(2, "Condimentos");
var Bebidas = new Categoria(3, "Bebidas");
```

```
Categorias.Add(Lacteos);
Categorias.Add(Condimentos);
Categorias.Add(Bebidas);

foreach(Categoria C in Categorias)
{
    Console.WriteLine($"Categoría: {C.ID}, {C.Nombre}");
}
```

Debido a que con **ArrayList** todos los elementos son tratados como objetos y se tiene que realizar conversiones implícitas y explicitas, trabajar con **ArrayList** no siempre genera un buen rendimiento. Vale la pena utilizar su contraparte colección genérica **List** que veremos posteriormente.

Para obtener más información acerca de la clase **ArrayList**, se recomienda visitar el siguiente enlace:

ArrayList Class
https://msdn.microsoft.com/en-us/library/system.collections.arraylist(v=vs.110).aspx

Utilizando Colecciones de tipo Diccionario

Las clases diccionario almacenan colecciones de parejas Llave/Valor. La clase diccionario más utilizada es el **Hashtable**. Cuando agregamos un elemento a una colección **Hashtable**, debemos especificar una llave y un valor. Tanto la llave como el valor pueden ser instancias de cualquier tipo, pero **Hashtable** convierte implícitamente la llave y el valor al tipo **Object**. Al recuperar los valores de la colección, se debe convertir el objeto a su tipo original.

El siguiente ejemplo, muestra como agregar y recuperar elementos de una colección **Hashtable**. En este caso, la llave es un tipo **string** y el valor es un tipo **int**.

```
// Crear una colección Hashtable.
Hashtable Ingredientes = new Hashtable();

 // Agregar algunos elementos llave/valor a la colección.
Ingredientes.Add("Leche", 12);
Ingredientes.Add("Azúcar", 7);
Ingredientes.Add("Pan", 1);
Ingredientes.Add("Frijoles", 11);

// Verificar si una llave existe.
if(Ingredientes.ContainsKey("Azúcar"))
{
    // Recuperar el valor asociado a la llave.
    Console.WriteLine($"El precio de Azúcar es {Ingredientes["Azúcar"]} ");
}
```

Las clases diccionario, como **Hashtable**, contienen dos colecciones enumerarles – las llaves y los valores. Se puede iterar sobre cualquiera de estas colecciones. En la mayoría de los casos, la iteración se realiza por la colección de llave para recuperar el valor asociado con cada llave.

El siguiente ejemplo, muestra como iterar sobre las llaves de la colección **Hashtable** y recupera el valor asociado con cada llave.

```
foreach (string Key in Ingredientes.Keys)
{
    Console.WriteLine($"Llave:{Key}, Valor:{Ingredientes[Key]}");
}
```

Para obtener más información acerca de la clase **Hashtable**, se recomienda visitar el siguiente enlace:

Hashtable Class
https://msdn.microsoft.com/en-us/library/system.collections.hashtable(v=vs.110).aspx

Realizando consultas sobre Colecciones

LINQ es una tecnología de consulta que se encuentra integrada en lenguajes .NET como Visual C#. LINQ permite utilizar una sintaxis de consulta declarativa estandarizada para consultar datos provenientes de diversas fuentes de datos tales como colecciones .NET, bases de datos SQL Server, Datasets de ADO.NET y documentos XML. Una consulta LINQ básica utiliza la siguiente sintaxis:

from <Nombre-la-variable-que-almacenará-un-elemento-a-la-vez-de-la-colección>

in <fuente-de-datos>

where <criterios de selección>

orderby <criterio-para-ordenar-los-datos>

select <nombres-de-variables>

Por ejemplo, supongamos que de nuestra colección **HashTable** quisiéramos mandar a la consola el nombre de aquellos productos cuyo precio sea mayor que 10 y ordenados por precio. Lo podríamos hacer de la siguiente forma:

```
// Seleccionar todos los ingredientes cuyo precio es mayor que 10
// ordenados por su precio de forma ascendente.
var MayoresQue10 =
    from string Ingrediente in Ingredientes.Keys
    where (int)Ingredientes[Ingrediente] > 10
    orderby Ingredientes[Ingrediente] ascending
    select Ingrediente;

// Imprimir el resultado
foreach (string Ingrediente in MayoresQue10)
{
    Console.WriteLine(Ingrediente);
}
```

Además de esta sintaxis de consulta básica, podemos invocar diversos métodos sobre el resultado de la consulta. Por ejemplo:

- El método **FirtsOrDefault** nos permite obtener el primer elemento de una colección o un valor predeterminado si la colección no contiene ningún elemento. Este método es útil si se tiene ordenado el resultado de la consulta.
- El método **LastOrdefault** nos permite obtener el último elemento de la colección o un valor predeterminado si la colección no contiene ningún elemento. Este método es útil si se tiene ordenado el resultado de la consulta.
- El método **Max** nos permite encontrar el elemento más grande de la colección.
- El método **Min** nos permite encontrar el elemento más pequeño de la colección.

Para obtener más información acerca de **LINQ**, se recomienda visitar el siguiente enlace:

LINQ Query Expressions (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/bb397676(v=vs.140)

Lección 3:
Manejando Eventos

Los eventos son mecanismos que permiten a los objetos notificar a otros objetos cuando algo ocurre. Por ejemplo, los controles de la interfaz de usuario de una aplicación Web o WPF generan eventos cuando el usuario interactúa con el control, tal como al dar clic en un botón. Nosotros podemos crear código para *suscribir* nuestro código a esos eventos y realizar alguna acción como respuesta a un evento.

Sin eventos, el código tendría que leer constantemente los valores del control para buscar si hay cambios en el estado que requieran de alguna acción. Esto sería una forma muy ineficiente de desarrollo de una aplicación.

En esta lección, aprenderemos a crear y lanzar eventos. Describiremos también la forma de suscribir nuestro código a eventos, así como la forma de manejar los eventos.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Crear eventos y delegados.
- Disparar eventos.
- Suscribir a eventos y eliminar la suscripción a eventos.

¿Qué es un Evento?

Los eventos son mecanismos que permiten a un Objeto o Clase notificar a otros Objetos o Clases cuando sucede algo relevante. El objeto o clase que envía (o genera) el evento, recibe el nombre de **Editor** y el Objeto o Clase que recibe (o maneja) el evento se denomina **Suscriptor**.

Por ejemplo, un objeto botón de comando en una aplicación Web o Windows puede notificar cuando el usuario hace clic en él. El botón notifica disparando su evento **Click**.

Un objeto **TextBox** puede notificar cuando su propiedad **Text** cambie, lanzando su evento **TextChanged**.

Los eventos tienen las propiedades siguientes:

- El Editor determina cuándo se produce un evento. El Suscriptor determina qué operación se realiza en respuesta al evento.
- Un evento puede tener varios Suscriptores. Un Suscriptor puede controlar varios eventos de varios Editores.
- No se invoca nunca a los eventos que no tienen suscriptores.
- Los eventos se utilizan normalmente para señalar acciones del usuario tal como hacer clic en un botón o seleccionar un menú en interfaces gráficas de usuario.
- Si un evento tiene varios suscriptores, de manera predeterminada los controladores de eventos se invocan sincrónicamente cuando se produce el evento, sin embargo, también es posible invocar de forma asincrónica los eventos. Los eventos se pueden utilizar para sincronizar subprocesos.
- En la biblioteca de clases base del .NET Framework, los eventos se basan en el delegado **EventHandler** y en la clase base **EventArgs**.

Escenario

Antes de mostrar cómo crear y utilizar Eventos de una clase, empecemos por plantear un escenario que nos permita ejemplificar dichos conceptos.

Supongamos que queremos crear una Clase que nos permita manejar operaciones de una cuenta bancaria. La Clase deberá poder realizar operaciones de retiro y deposito.

El código de la clase podría quedar de la siguiente forma:

```
class CuentaBancaria
{
    private decimal SaldoActual;
    public decimal Retirar(decimal importe)
    {
        if (SaldoActual >= importe)
        {
            SaldoActual -= importe;
        }
        return SaldoActual;
    }
    public decimal Depositar(decimal importe)
    {
        SaldoActual += importe;
        return SaldoActual;
    }
}
```

El siguiente ejemplo, muestra la forma en que podemos consumir la clase **CuentaBancaria**.

```
CuentaBancaria Cuenta =
    new CuentaBancaria();
var SaldoActual = Cuenta.Depositar(1000);
SaldoActual = Cuenta.Retirar(200);
```

Si observamos el código del método **Retirar** de la clase **CuentaBancaria**, podemos notar que cuando ya no hay saldo disponible simplemente no se realiza el retiro y el saldo permanece sin cambios.

¿Qué podríamos hacer para que la clase **CuentaBancaria** pueda ***notificar*** cuando no hay saldo disponible?

Podríamos desear que cuando la clase **CuentaBancaria** detecte que ya no hay saldo disponible, pudiera invocar a un método enviándole un mensaje apropiado.

Debido a que lo que deseamos es que la Clase **CuentaBancaria** ***Notifique*** a otra clase que ya no hay saldo disponible para retiro y recordando la definición de que un evento es un mecanismo que permite a una clase ***notificar*** a otra clase cuando suceda algo relevante, podemos entonces concluir que lo que necesitamos es definir un evento en la clase **CuentaBancaria**.

Delegados

Continuando con el escenario planteado anteriormente, deseamos que la clase **CuentaBancaria** notifique al código cliente cuando ya no haya saldo por retirar. Específicamente, queremos que la clase **CuentaBancaria** pueda ejecutar un método que podría encontrase en otra clase. Para que esto sea posible, la clase **CuentaBancaria** necesita conocer la dirección de memoria en la que se encuentra ubicado el método a ejecutar.

Así como existe el tipo **int** para almacenar números enteros, el tipo **string** para almacenar cadenas de caracteres o el tipo **DateTime** que almacena fechas y horas, en C# podemos definir un tipo especial de dato que nos permita almacenar **direcciones** de métodos. Este tipo especial de dato que nos permite almacenar direcciones de métodos recibe el nombre de **Delegado**.

Un tipo delegado nos permite declarar variables que pueden almacenar una o más direcciones de métodos. Un tipo delegado puede considerarse como un apuntador de direcciones de métodos en memoria.

Al definir un tipo delegado, debemos especificar el tipo de valor devuelto y la lista de parámetros de los métodos cuya dirección podrá almacenar el tipo Delegado. Un tipo delegado solo podrá almacenar direcciones de métodos que cumplan con la especificación declarada al definir el tipo delegado.

El siguiente ejemplo, define un tipo delegado que podrá almacenar direcciones de métodos que no devuelvan valor y requieran un tipo **string** como parámetro. Nótese que, para definir un tipo delegado, utilizamos la palabra clave **delegate**.

```
public delegate void ErrorHandler(string info);
```

Con el tipo delegado ya definido, podemos declarar una variable o propiedad de tipo **ErrorHandler** en la clase **CuentaBancaria** para permitir almacenar direcciones de métodos que no devuelvan valor y que requieran un solo parámetro de tipo **string**.

```
public ErrorHandler Apuntador { get; set; }
```

Debido a que las variables de tipo delegado representan direcciones de métodos, podemos invocar esas direcciones a través de estas variables.

El siguiente ejemplo, ejecuta el método o métodos cuya dirección se almacena a través de la propiedad **Apuntador** para notificar que ya no hay saldo disponible o que el saldo está en cero.

```
public decimal Retirar(decimal importe)
{
    if (SaldoActual >= importe)
    {
        SaldoActual -= importe;
        if (SaldoActual == 0)
```

```
            {
                if (Apuntador != null) // ¿Hay una dirección de un método a notificar?
                {
                    Apuntador("El Saldo es cero"); // Ejecutar el método.
                }
            }
        }
        else
        {
            if (Apuntador != null) // ¿Hay una dirección de un método a notificar?
            {
                Apuntador("No hay saldo disponible"); // Ejecutar el método.
            }
        }
        return SaldoActual;
    }
```

El siguiente ejemplo muestra la forma en que podemos proporcionar una dirección de un método a la clase **CuentaBancaria**.

```
            CuentaBancaria Cuenta =
                new CuentaBancaria();

            // Proporcionar la dirección del método NotificarSaldo
            Cuenta.Apuntador = NotificarSaldo;
```

El siguiente ejemplo, muestra el código del método NotificarSaldo.

```
        void NotificarSaldo(string mensaje)
        {
            // .....
        }
```

La propiedad **Apuntador** no podrá almacenar direcciones de métodos que no cumplan la especificación del delegado.

El siguiente código no compilaría ya que el delegado define un solo parámetro de tipo **string** pero el método **NotificarSaldo** define 2 parámetros.

```
        void NotificarSaldo(string mensaje, int severidad)
        {
            // .....
        }
```

Como mencionamos previamente, un tipo delegado puede almacenar una o más direcciones y todas esas direcciones serán ejecutadas al invocar el delegado.

El siguiente código agrega una nueva dirección de un método a la propiedad Apuntador. Ambos métodos serán ejecutados al invocar el delegado.

```
            Cuenta.Apuntador += NotificarError;
```

Eventos

Como comentamos previamente, los eventos son mecanismos que permiten a un Objeto o Clase notificar a otros Objetos o Clases cuando sucede algo relevante. El objeto o clase que envía (o genera) el evento, recibe el nombre de **Editor**. En nuestro caso el Editor es la clase **CuentaBancaria**. Esta clase expondrá un **Evento** para notificar al código cliente cuando ya no haya saldo.

El Objeto o Clase que recibe (o maneja) el evento se denomina **Suscriptor**.

Cuando creamos un evento en una estructura o una clase, necesitamos una manera para permitir a otro código suscribirse a ese evento. En Visual C#, se logra esto mediante la creación de un delegado. Al definir un evento, debemos asociar dicho evento con un delegado.

Una buena práctica relacionada con la creación de Eventos es crear un Delegado con la siguiente firma:

```
public delegate void ErrorEventHandler(object sender, ErrorEventArgs e);
```

El primer parámetro representa al objeto que genera el evento. El segundo parámetro representa los datos del evento, esto es, cualquier información que deseamos proporcionar a los consumidores. El segundo parámetro debe ser una instancia de la clase **EventArgs** o una instancia de una clase derivada de **EventArgs.**

El siguiente ejemplo, define la clase **ErrorEventArgs** que permitirá notificar al código del suscriptor acerca de un error. La clase **ErrorEventArgs** permite proporcionar un mensaje de error y la severidad del error.

```
public class ErrorEventArgs : EventArgs
{
    public string Mensaje { get; set; }
    public byte Severidad { get; set; }
}
```

El siguiente ejemplo muestra la definición de un evento llamado **Error** que permitirá notificar al suscriptor cuando no haya saldo disponible. El evento podrá invocar direcciones de métodos que satisfagan la especificación del delegado **ErrorEventHandler**.

```
public event ErrorEventHandler Error;
```

Después de definir un delegado y un evento, podemos escribir el código que lanzará el evento cuando se cumplan ciertas condiciones. Cuando se lanza el evento, el delegado asociado con el evento invocará cualquier método que se haya suscrito a dicho evento.

Lanzar un evento es similar a invocar un delegado. Primero, debemos verificar si el evento es nulo. El evento será nulo si ningún código está suscrito actualmente a él.

El siguiente ejemplo, muestra la forma de lanzar el evento **Error** para notificar que ya no hay saldo disponible o que el saldo está en cero.

```
public decimal Retirar(decimal importe)
{
    if (SaldoActual >= importe)
    {
        SaldoActual -= importe;
        if (SaldoActual == 0)
        {
            if (Error != null) // ¿Hay código suscrito al evento?
            {
                var Editor = this; // El Objeto actual es quien lanza el evento.
                var DatosDelEvento =
                    new ErrorEventArgs();
                DatosDelEvento.Mensaje = "El Saldo es cero";
                DatosDelEvento.Severidad = 0;

                Error(Editor, DatosDelEvento); // Lanzar el evento.
            }
        }
    }
    else
    {
        if (Error != null) // ¿Hay código suscrito al evento?
        {
            var Editor = this; // El Objeto actual es quien lanza el evento.
            var DatosDelEvento =
                new ErrorEventArgs();
            DatosDelEvento.Mensaje = "No hay saldo disponible";
            DatosDelEvento.Severidad = 10;

            Error(Editor, DatosDelEvento); // Lanzar el evento.
        }
    }
    return SaldoActual;
}
```

Suscribiendo a Eventos

Si deseamos controlar un evento en el código cliente, necesitamos hacer dos cosas:

1. Debemos crear un método que coincida con la definición del delegado del evento. Este método recibe el nombre de controlador del evento o *Event Handler*.
2. Utilizamos el operador de asignación adicional (+=) para suscribir el método manejador de evento al evento mismo.

El siguiente ejemplo muestra la definición de un método para manejar el evento **Error** de la clase **CuentaBancaria**.

```
void Cuenta_Error(object sender, ErrorEventArgs e)
{
    Console.WriteLine($"Error:{e.Mensaje}, Severidad:{e.Severidad}");
}
```

El siguiente ejemplo, muestra la forma de suscribir el método **Cuenta_Error** como manejador del evento **Error**.

```
CuentaBancaria Cuenta = new CuentaBancaria();

// Proporcionar la dirección del manejador del evento Error.
Cuenta.Error += Cuenta_Error;
```

Un suscriptor se puede suscribir más de una vez a un evento. El siguiente ejemplo, muestra la forma de suscribir el método **Notificar_Error** como otro manejador del evento **Error**.

```
Cuenta.Error += Notificar_Error;
```

Para cancelar una suscripción a un evento, utilizamos el operador de asignación de sustracción (-=) para remover el método manejador de evento.

El siguiente ejemplo, elimina la suscripción del manejador de evento **Notificar_Error**.

```
Cuenta.Error -= Notificar_Error;
```

Trabajando con Eventos en XAML

En las aplicaciones XAML, podemos interactuar con el usuario mediante la definición de manejadores de eventos para responder a las distintas acciones que realice el usuario sobre los elementos de la interfaz de usuario.

El siguiente código XAML muestra como suscribir el método **btnSaluda_Click** como manejador del evento **Click** del botón **btnSaluda**.

```
<Button x:Name="btnSaluda" Content="Saluda!"
        Click="btnSaluda_Click"/>
```

Visual Studio procesa el código XAML para crear el código que suscribe el método manejador de evento con el evento **Click** del botón de comando.

El código XAML anterior genera el siguiente código C#.

```
this.btnSaluda.Click += new System.Windows.RoutedEventHandler(this.btnSaluda_Click);
```

El siguiente ejemplo, muestra el código del manejador de evento **btnSaluda_Click**.

```
private void btnSaluda_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hola, mundo!");
}
```

Introducción a C#

Módulo 4: Creando Clases e implementando colecciones de Tipos Seguros (Type-safe collections)

Acerca del módulo

Las clases nos permiten crear nuestros propios tipos personalizados, independientes y reutilizables. Las Interfaces nos permiten definir un conjunto de elementos que las clases deben implementar para garantizar la compatibilidad con los consumidores de las clases.

En este módulo, aprenderemos a utilizar las interfaces y clases para definir y crear nuestros propios tipos personalizados, reutilizables. También aprenderemos a crear y utilizar, colecciones enumerables de tipos seguros.

Objetivos

Al finalizar este módulo, los participantes contarán con las habilidades y conocimientos para:

- Crear y utilizar Clases personalizadas.
- Definir e implementar Interfaces personalizadas.
- Utilizar Tipos Genéricos para implementar colecciones de tipos seguros (Type-safe Collections).

Los temas que se cubren en este módulo son:

- Lección 1: Creando Clases.
- Lección 2: Definiendo e implementando Interfaces.
- Lección 3. Implementando colecciones de tipos seguros (Type-safe Collections).

Lección 1:
Creando Clases

En Visual C#, podemos crear Clases para definir nuestros propios tipos personalizados de datos. Como una construcción de programación, la Clase es el elemento central de la Programación Orientada a Objetos en Visual C#. Esto nos permite encapsular los comportamientos y características de cualquier entidad lógica en una forma reutilizables y extensible.

En esta lección, aprenderemos a crear, utilizar y probar la funcionalidad de las Clases de nuestras aplicaciones.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Crear Clases.
- Instanciar Clases para crear Objetos.
- Utilizar Constructores para establecer valores o para ejecutar lógica al momento de instanciar la Clase.
- Explicar la diferencia entre ***Tipos Referencia*** y ***Tipos Valor***.
- Crear Clases y Miembros estáticos.
- Describir el proceso de alto nivel para probar la funcionalidad de las Clases.

Creando Clases y sus miembros

En Visual C#, podemos definir nuestros propios tipos de datos personalizados mediante la creación de clases. La clase es el elemento central de la programación orientada a objetos en Visual C#. La Clase define los comportamientos y características de cualquier entidad lógica de una forma reutilizable y extensible. Los comportamientos y características son conocidos como miembros de la Clase y son representados mediante la definición de *Métodos*, *Campos*, *Propiedades* y *Eventos* dentro de la clase.

Declarando una Clase

Utilizamos la palabra clave **class** para declarar una clase como se muestra en el siguiente ejemplo.

```
public class Automovil
{
    // Aquí van los Métodos, Campos, Propiedades y Eventos.
}
```

La palabra clave **class** es precedida por un modificador de acceso, tal como **public** en nuestro ejemplo, que específica desde donde se puede utilizar el tipo. Podemos utilizar los siguientes modificadores de acceso en la declaración de una clase.

Modificador de acceso	Descripción
public	La clase está disponible para el código que se ejecuta en cualquier Assembly que hace referencia al Assembly en el que está contenida la clase.
internal	La clase está disponible para cualquier código dentro del mismo Assembly, pero no está disponible para el código de otro Assembly. Este es el valor por defecto si no se especifica un modificador de acceso.
private	La clase sólo está disponible para el código dentro de la clase que lo contiene. Sólo se puede utilizar el modificador de acceso **private** con clases anidadas.

Agregando Miembros a una Clase

Podemos utilizar los Campos y Propiedades para definir las características de una Clase tales como Modelo, Color o Marca en la clase **Automovil**. Podemos crear Métodos para representar las cosas que un **Automovil** puede realizar tal como Arrancar, Detener o Caminar. Finalmente, podemos definir eventos para representar las acciones que pudieran requerir nuestra atención tales como llenar el auto con gasolina cuando se esté agotando.

El siguiente ejemplo, muestra algunos miembros de la clase **Automovil**.

```
public class Automovil
{
    // Campo
    public string Modelo;

    // Propiedades
    public string Color { get; set; }
    public string Marca { get; set; }

    // Métodos
    public void Arrancar()
    {
        //...
    }
    public void Detener()
    {
        //...
    }
    public void Caminar()
    {
        //...
    }

    // Evento
    public event EventHandler GasolinaAgotada;
}
```

Instanciando Clases

Una clase es sólo una plantilla para un Tipo de dato. Para utilizar los comportamientos y características que definimos dentro de una clase, es necesario crear *instancias* de la clase. Una instancia de una clase es llamada ***Objeto***.

Para crear una nueva instancia de una clase, se utiliza la palabra clave **new** como se muestra en el siguiente ejemplo**.**

```
Automovil auto = new Automovil();
```

Cuando creamos una instancia de una clase de esta manera, realmente estamos realizando dos cosas:

- Estamos creando un nuevo objeto en memoria basado en el tipo **Automovil**.
- Se está creando una referencia a un objeto llamado **auto** que hace referencia al nuevo objeto **Automovil**.

Cuando creamos la referencia al objeto, en lugar de especificar explícitamente el tipo **Automovil**, podemos permitir que el compilador deduzca el tipo de objeto en tiempo de compilación. Esto es conocido como ***inferencia de tipos***. Para utilizar la inferencia de tipos, podemos crear la referencia a un objeto mediante la palabra clave **var** como se muestra en el siguiente ejemplo.

```
var Auto = new Automovil();
```

En este caso, el compilador no sabe de antemano el tipo de la variable **Auto**. Cuando la variable **Auto** es inicializada como una referencia a un objeto **Automovil**, el compilador deduce que el tipo de **Auto** es **Automovil**.

Utilizando la inferencia de tipos de esta manera, no causa ningún cambio en la forma en que se ejecuta la aplicación, es simplemente un atajo para evitar escribir el nombre de la clase dos veces. En algunas circunstancias, la inferencia de tipos puede hacer que nuestro código sea más fácil de leer, mientras que, en otras circunstancias, puede hacer que el código sea más confuso. Como regla general, consideremos el uso de inferencia de tipos cuando el tipo de una variable es absolutamente claro.

Después de haber instanciado el objeto, podemos utilizar cualquiera de los miembros --métodos, campos, propiedades y eventos-- que hayamos definido dentro de la clase como se muestra en el siguiente ejemplo.

```
var auto = new Automovil();
auto.Color = "Verde";
auto.Marca = "VW";
auto.Modelo = "Europa";
auto.Arrancar();
```

Este enfoque para invocar a los miembros de una variable de instancia es conocido como notación por puntos.

Podemos escribir el nombre de la variable, seguido de un punto, seguido por el nombre del miembro. La característica **IntelliSense** en Visual Studio nos mostrará los nombres de los miembros al escribir un punto después de una variable.

Utilizando Constructores

Hemos visto que la sintaxis para instanciar una clase, por ejemplo, **new Automovil()**, es similar a la sintaxis para invocar a un método. Esto es porque cuando instanciamos una clase, estamos invocando a un método especial llamado ***constructor***. Un constructor es un método especial en la clase que tiene el mismo nombre que la clase.

Los Constructores se utilizan a menudo para especificar los valores iniciales o predeterminados para los miembros dentro del nuevo objeto como se muestra en el siguiente ejemplo.

```
public Automovil()
{
    Modelo = "Europa";

}
```

Un constructor que no tiene parámetros se conoce como el *constructor predeterminado*. Este constructor es invocado cuando alguien crea una instancia de la clase sin proporcionar ningún argumento.

Una regla importante relacionada con los constructores, es que todas las clases deben tener un constructor. Si nosotros no incluimos un constructor en la clase, el compilador de Visual C# agregará automáticamente un constructor público predeterminado vacío a la clase compilada.

Una segunda regla relacionada con los constructores, es que el constructor debe invocar al constructor de su clase base. Si no lo hacemos nosotros, el compilador agregará el código para invocar al constructor predeterminado, esto es, aquel que no tiene parámetros.

En muchos casos, es útil para los consumidores de nuestra clase, poder especificar los valores iniciales para los miembros de datos cuando la clase es instanciada. Por ejemplo, cuando alguien crea una nueva instancia de **Automovil**, podría ser útil si se puede especificar el modelo del **Automovil**. Nuestra clase puede incluir varios constructores con diferentes firmas que permitan a los consumidores proporcionar diferentes combinaciones de información al momento de crear una instancia de la clase.

El siguiente ejemplo muestra como agregar múltiples constructores a la clase.

```
public class Automovil
{
    public Automovil()
    {
        Modelo = "Europa";
    }

    public Automovil(string modelo)
    {
```

```
            this.Modelo = modelo;
        }

        public Automovil(string modelo, string color)
        {
            this.Modelo = modelo;
            this.Color = color;
        }

        public Automovil(string modelo, string color, string marca)
        {
            this.Modelo = modelo;
            this.Color = color;
            this.Marca = marca;
        }

        public string Modelo;
        public string Color { get; set; }
        public string Marca { get; set; }

        public void Arrancar()
        {
            //...
        }
        public void Detener()
        {
            //...
        }
        public void Caminar()
        {
            //...
        }
        public event EventHandler GasolinaAgotada;

    }
```

Los consumidores, pueden utilizar cualquiera de los constructores para crear instancias de la clase dependiendo de la información que tengan disponible en ese momento. Por ejemplo:

```
            var Auto = new Automovil();
            var Chevy = new Automovil("Chevy");
            var VWRojo = new Automovil("VW", "Rojo");
            var Jetta = new Automovil("Europa", "Plata", "VW");
```

Tipos Referencia y Tipos Valor

Ahora que sabemos cómo crear e instanciar clases, aprenderemos acerca de las diferencias entre clases y estructuras.

Una estructura es un ***tipo valor***. Esta definición aplica a los tipos struct integrados, tales como **int** y **bool**, así como para las estructuras que definimos nosotros mismos. Un tipo valor contiene su dato directamente. En otras palabras, cuando interactuamos con un tipo de valor, estamos interactuando directamente con el dato que contiene en memoria. La memoria donde se almacenan los tipos valor se conoce como ***Stack*** o ***Pila***.

A diferencia de una estructura, una clase define un ***tipo referencia***. Cuando creamos un objeto instanciando una clase, estamos creando un tipo referencia. El objeto mismo es almacenado en memoria, una memoria conocida como ***Heap*** o ***montón***, pero interactuamos con el objeto a través de una referencia de objeto. Todo lo que la referencia de objeto hace es apuntar al objeto en memoria. La referencia de objeto que es una dirección de memoria, es almacenada en la dirección donde está ubicada la variable que tiene la instancia de la clase.

Para acceder al dato de un tipo valor, lo hacemos directamente en la memoria Stack y para acceder al dato de un tipo referencia, primero obtenemos del Stack la dirección del objeto y con esa referencia accedemos al dato del objeto en la memoria Heap.

Los tipos valor y tipos referencia se comportan de manera diferente. Si copiamos un tipo valor desde una variable a otra, estamos copiando el dato que la variable contiene y creando una nueva instancia de ese dato en memoria. Si modificamos los datos de una de esas variables, no afectamos al valor de la otra.

Por el contrario, si copiamos una referencia de objeto de una variable a otra, lo que estamos haciendo es copiar la referencia al objeto, en otras palabras, estamos copiando la dirección del objeto, no estamos creando un segundo objeto en memoria, ambas variables apuntan al mismo objeto. Por lo que, si modificamos un miembro, estaremos modificando el único miembro al que apuntan las dos variables de tipo referencia.

Boxing y Unboxing

En algunos casos es posible que tengamos que convertir los tipos valor a tipos referencia y viceversa.

Por ejemplo, algunas clases colección sólo aceptan tipos referencia, aunque con la llegada de los tipos Genéricos esto prácticamente ya no es un problema, sin embargo, aun debemos estar conscientes de ello ya que un concepto fundamental de Visual C# es que se puede tratar cualquier tipo como un objeto.

El proceso de conversión de un tipo valor a un tipo referencia es llamado ***boxing***. Para "encajonar" una variable, simplemente la asignamos a una referencia de objeto.

```
int i = 5;
object o = i;
```

El proceso ***boxing*** es implícito. Cuando asignamos un tipo valor a una referencia de objeto, el compilador de Visual C# crea automáticamente un objeto para contener el valor y lo almacena en memoria. Si se copia la referencia de objeto, la copia apuntará al mismo objeto almacenado en la memoria.

El proceso de conversión de un tipo referencia a un tipo valor es llamado ***unboxing***. A diferencia del proceso ***boxing***, para "desencajonar" un tipo valor, debemos convertir explícitamente la variable a su tipo original.

```
int j = (int)o;
```

Creando Clases y Miembros Estáticos

En algunos casos, es posible que deseemos crear una clase exclusivamente para encapsular alguna funcionalidad útil, en vez de representar una instancia de algún objeto. Por ejemplo, supongamos que deseamos crear un conjunto de métodos que realicen conversiones de unidades de medida como, por ejemplo, de Libras a Kilogramos o de Millas a Kilómetros entre otras conversiones. No tendría sentido si tuviéramos que crear una instancia de una clase con el fin de utilizar esos métodos ya que no es necesario almacenar o recuperar algún dato especifico de una instancia. De hecho, el concepto de una instancia no tiene sentido en este caso.

En escenarios como este, podemos crear una ***clase estática***. Una clase estática es una clase que no puede ser instanciada. Para crear una clase estática, se utiliza la palabra clave **static** como se muestra en el siguiente ejemplo.

```
static class Conversiones
{
}
```

Cualquier miembro dentro de la clase también debe utilizar la palabra clave **static**, esto es, también debe ser estático. Para declarar un miembro estático se utiliza la palabra clave **static** antes del tipo devuelto del miembro como se muestra en el siguiente ejemplo.

```
static class Conversiones
{
    public static double ConvertirLibrasAKilogramos(double libras)
    {
        return libras * 0.45359237;
    }
}
```

Para invocar a un método en una clase estática, invocamos al método sobre el nombre de la clase misma en lugar del nombre de una instancia, como se muestra en el siguiente ejemplo.

```
        var Kilos = Conversiones.ConvertirLibrasAKilogramos(200);
```

Miembros estáticos

Las clases no estáticas pueden incluir miembros estáticos. Esto es útil cuando algunos comportamientos y características se relacionan con la instancia (miembros de instancia), mientras que otros comportamientos y características se relacionan con el tipo mismo (miembros estáticos). Los Métodos, Campos, Propiedades y Eventos, todos ellos pueden ser declarados como estáticos. Las propiedades estáticas se utilizan a menudo para devolver datos que son comunes a todas las instancias o para realizar un seguimiento de cuántas instancias de una clase se han creado. Los métodos estáticos frecuentemente son utilizados para proporcionar funcionalidad que se relaciona de alguna manera con el tipo, tales como funciones de comparación.

El siguiente ejemplo muestra un miembro estático en una clase no estática.

```
class Conversiones
{
    public static double ConvertirLibrasAKilogramos(double libras)
    {
        return libras * 0.45359237;
    }
}
```

Independientemente de cuantas instancias de la clase existan, hay solo una instancia de un miembro estático.

Probando el funcionamiento de las Clases

Las clases a menudo representan unidades autónomas de funcionalidad. En muchos casos, desearemos probar la funcionalidad de nuestras clases por separado antes de integrarlas con otras clases de la aplicación.

Para probar la funcionalidad de forma aislada, creamos una prueba unitaria. Una prueba unitaria presenta al código bajo prueba con entradas conocidas, realiza una acción sobre el código bajo prueba (por ejemplo, invocando a un método) y luego verifica que el resultado de la operación sea como era esperado. De esta manera, la prueba unitaria representa un contrato que el código debe cumplir. Sin embargo, al cambiar la implementación de una clase o método, la prueba unitaria asegura que nuestro código siempre devuelva salidas particulares en respuesta a entradas particulares.

Por ejemplo, consideremos el caso donde se crea una clase simple para representar a un cliente. La clase Cliente incluye un método llamado **ObtenerEdad** que devuelve la edad actual del cliente en años.

```
public class Cliente
{
    public DateTime FechaDeNacimiento { get; set; }
    public int ObtenerEdad()
    {
        TimeSpan Diferencia = DateTime.Now.Subtract(FechaDeNacimiento);
        int Edad = (int)(Diferencia.Days / 365.25);
        return Edad;
    }

}
```

En este caso, es posible que deseemos crear una prueba unitaria que asegure que el método **ObtenerEdad** se comporta como esperamos. Por lo tanto, el método de prueba debe crear una instancia de la clase **Cliente**, especificar la fecha de nacimiento y luego verificar que el método **ObtenerEdad** devuelva la edad correcta en años. Dependiendo el Framework de pruebas unitarias que utilicemos, el método de prueba podría ser similar al siguiente.

```
    [TestMethod]
    public void TestCliente()
    {
        // Organizamos (Arrange)
        DateTime FechaDeNacimiento = DateTime.Today;
        FechaDeNacimiento = FechaDeNacimiento.AddDays(7);
        FechaDeNacimiento = FechaDeNacimiento.AddYears(-24);
        Cliente c = new Cliente();
        c.FechaDeNacimiento = FechaDeNacimiento;
        // Los 24 años del cliente serán dentro de 7 días
        // por lo que la edad en años debería ser 23.
        int EdadEsperada = 23;
```

```
        // Actuar (Act)
        int EdadActual = c.ObtenerEdad();

        // Afirmar (Assert)
        // Falla si la prueba de la edad actual y la edad esperada son diferentes.
        Assert.IsTrue(EdadActual == EdadEsperada,
            "La edad no se está calculando correctamente");
    }
```

Podemos notar que los métodos de prueba unitaria se dividen en tres fases conceptuales:

- **La fase Organizar (Arrange)**. En esta fase, se crean las condiciones para la prueba. Creamos una instancia de la clase que queremos probar y configuramos cualquier valor de entrada que la prueba requiera.
- **La fase Actuar (Act).** En esta fase, se realiza la acción que se desea probar.
- **La fase Afirmar (Assert).** En esta fase, se verifica el resultado de la acción. Si el resultado no fue lo esperado, la prueba falla.

El método **Assert.IsTrue** es parte del Framework de pruebas unitarias de Microsoft que se incluye en Visual Studio. Este método particular dispara una excepción si la condición especificada no se evalúa como **true**. Sin embargo, los conceptos descritos aquí son comunes a todos los Frameworks de pruebas unitarias.

Lección 2:
Definiendo e implementando Interfaces

Una Interface es muy similar a una Clase, pero sin la implementación. La Interface especifica un conjunto de características y comportamientos mediante la definición de firmas de métodos, propiedades, eventos e indizadores, sin especificar la forma en que son implementados esos miembros. Cuando una clase implementa una Interface, la clase proporciona una implementación de cada miembro de la Interface. Mediante la implementación de la Interface, la clase garantiza que proporcionará toda la funcionalidad especificada por la Interface.

En esta lección, aprenderemos a definir e implementar Interfaces.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Explicar el propósito de las Interfaces en Visual C#.
- Crear Interfaces.
- Crear clases que implementen una única Interface.
- Crear clases que implementen múltiples Interfaces.
- Implementar la Interface **IComparable**.
- Implementar la Interface **IComparer**.

Introducción a las Interfaces

En Visual C #, una Interface especifica un conjunto de características y comportamientos mediante la definición de métodos, propiedades, eventos e indizadores. Una Interface es muy similar a una clase, pero sin una implementación. La Interface solo especifica la firma de los miembros, pero no especifica la forma en que deben implementarse esos miembros. De hecho, el detalle de la implementación no debe importar a los consumidores.

Una Interface no puede ser instanciada debido a que no contiene la implementación de los miembros, sólo sus definiciones.

Las clases y estructuras pueden implementar una Interface. Cuando las clases o estructuras tengan que implementar una Interface, deben implementar cada uno de sus miembros.

Los nombres y firmas de los miembros implementados deben ser iguales a los nombres y firmas definidos en la Interface.

Podemos pensar que una Interface es como un contrato. Mediante la implementación de una Interface particular, una clase garantiza a los consumidores que proporcionará una funcionalidad específica a través de los miembros específicos, incluso aunque la implementación actual no sea parte del contrato.

Definiendo Interfaces

La sintaxis para definir una Interface es similar a la sintaxis para definir una clase. Podemos utilizar la palabra clave ***interface*** para definir una interface de la siguiente forma:

```
interface IEmpleado
{
}
```

Por convención de programación, los nombres de la Interfaces deben empezar con una letra "I" mayúscula.

Al igual que en una declaración de clase, una declaración de interface puede incluir un modificador de acceso. Los modificadores de acceso que podemos utilizar para definir interfaces son ***public*** para hacer disponible la Interface en cualquier código de cualquier ensamblado e ***internal*** para hacer que la Interface esté disponible para cualquier código del mismo ensamblado pero que no esté disponible al código de un ensamblado distinto. ***internal*** es el valor predeterminado si no se especifica un modificador de acceso.

Una Interface define la firma de los miembros, pero no incluye ningún detalle de la implementación. Las interfaces pueden incluir Propiedades, Métodos, Eventos e Indizadores.

Para definir una propiedad, especificamos el nombre de la propiedad, el tipo de la propiedad y los descriptores de acceso de la propiedad como se muestra en el siguiente ejemplo.

```
interface IEmpleado
{
    string Nombre { get; set; }
    string Apellidos { get; set; }
    decimal SueldoDiario { get; }
}
```

La sintaxis para definir una propiedad es similar a la sintaxis de las propiedades automáticamente implementadas. La diferencia aquí es que el compilador no realizará ninguna implementación.

Los miembros de la interfaz no incluyen modificadores de acceso. El propósito de la interface, es definir los miembros que una clase que la implemente debe exponer a los consumidores y, por lo tanto, estos miembros son públicos y no debemos especificar algún modificador de acceso.

Para definir un método, se especifica el tipo de valor devuelto, el nombre del método y los parámetros. De ninguna manera se proporciona la implementación del método.

```
interface IEmpleado
{
    string Nombre { get; set; }
    string Apellidos { get; set; }
    decimal SueldoDiario { get; }
    decimal CalcularPagoMensual(byte diasTrabajados);
```

```
}
```

Como podemos notar del ejemplo anterior, la definición de un método no incluye el cuerpo del método.

Para definir un evento, se utiliza la palabra clave ***event***, seguido por el tipo delegado asociado al evento y el nombre del evento como se muestra en el siguiente ejemplo.

```
interface IEmpleado
{
    string Nombre { get; set; }
    string Apellidos { get; set; }
    decimal SueldoDiario { get; }
    decimal CalcularPagoMensual(byte diasTrabajados);
    event EventHandler AlDetectarAusenciaDePago;
}
```

Para definir un indizador, se especifica el tipo de valor devuelto y los descriptores de acceso como se muestra en el siguiente ejemplo.

```
interface IEmpleado
{
    string Nombre { get; set; }
    string Apellidos { get; set; }
    decimal SueldoDiario { get; }
    decimal CalcularPagoMensual(byte diasTrabajados);
    event EventHandler AlDetectarAusenciaDePago;
    bool this[byte index] { get; set; }
}
```

Las interfaces no pueden incluir a miembros que se relacionen con la funcionalidad interna de una clase tales como Campos, Constantes, Operadores y Constructores.

Implementando Interfaces

Las Interfaces no se heredan, las Interfaces se implementan. Para crear una clase que implemente una interface, debemos de agregar dos puntos a la declaración de la clase y, a continuación, el nombre de la interface como se muestra en el siguiente ejemplo.

```
class Gerente : IEmpleado
{
}
```

Dentro de la clase, debemos proporcionar una implementación para cada miembro de la Interface. La clase puede incluir a miembros adicionales que no estén definidos por la Interface. De hecho, la mayoría de las clases incluyen miembros adicionales porque generalmente la clase extiende la interface. Sin embargo, aunque la clase puede agregar miembros adicionales, no puede omitir la implementación de ninguno de los miembros. La manera de implementar a los miembros de la interface no importa, siempre y cuando las implementaciones tengan las mismas firmas y los mismos tipos de valor devuelto como son definidos en la interface.

El siguiente ejemplo, muestra la implementación de los miembros de la interface ***IEmpleado***.

```
class Gerente : IEmpleado
{
     public string Nombre { get; set; }
     public string Apellidos { get; set; }
     public decimal SueldoDiario
     {
         get{ return 1000; }
     }

     // Miembro adicional
     public decimal PrestacionesGerenciales { get; set; }

     public decimal CalcularPagoMensual(byte diasTrabajados)
     {
         return diasTrabajados * SueldoDiario + PrestacionesGerenciales;
     }

     public event EventHandler AlDetectarAusenciaDePago;

     public bool this[byte index]
     {
         get
         {
             return true;
         }
         set
         {
             // Almacenar el valor en la base de datos
         }
     }
}
```

El siguiente ejemplo muestra otra implementación de la interface ***IEmpleado***.

```
class Oficinista : IEmpleado
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public decimal SueldoDiario
    {
        get { return 1000; }
    }

    // Miembro adicional
    public decimal Deducciones { get; set; }

    public decimal CalcularPagoMensual(byte diasTrabajados)
    {
        return diasTrabajados * SueldoDiario - Deducciones;
    }

    public event EventHandler AlDetectarAusenciaDePago;

    public bool this[byte index]
    {
        get
        {
            return true;
        }
        set
        {
            // Almacenar el valor en la base de datos
        }
    }

}
```

Polimorfismo e Interfaces

Cuando trabajamos con Interfaces, no podemos dejar de mencionar uno de los pilares de la programación orientada a objetos que es el ***Polimorfismo***. En términos de Interfaces, el polimorfismo manifiesta que podemos representar una instancia de una clase como una instancia de cualquier Interface que la clase implementa.

El siguiente ejemplo, muestra que podemos utilizar el tipo interface y el tipo de la clase que implementa la interface de manera intercambiable.

```
Gerente g = new Gerente();
IEmpleado ie = new Gerente();
```

En el ejemplo anterior, una instancia de la clase ***Gerente*** es representada como si fuera una instancia de la Interface ***IEmpleado***.

El siguiente ejemplo, muestra que cuando convertimos una clase ***Gerente*** al tipo ***IEmpleado***, utilizamos una conversión implícita debido a que sabemos que la clase ***Gerente*** debe incluir todos los miembros de la interface.

```
Gerente g = new Gerente();
IEmpleado g3 = g;
```

Por el contrario, cuando convertimos un tipo interface a un tipo de la clase que implementa la interface, debemos realizar una conversión explicita debido a que la clase puede incluir miembros que no son definidos en la Interface.

```
g = (Gerente) g3; // o
g = g3 as Gerente;
```

Una variable declarada de tipo ***IEmpleado*** que almacene una instancia del tipo ***Gerente***, únicamente expone miembros definidos en la interface ***IEmpleado*** y no expone el miembro adicional ***PrestacionesGerenciales*** definido en la clase **Gerente**. Sin embargo, una variable declarada de tipo ***Gerente*** que almacene una instancia del tipo ***Gerente***, si la incluye.

El Polimorfismo por Interface, puede ayudar a aumentar la flexibilidad y modularidad del código. Supongamos que tenemos varias clases que implementan la interface ***IEmpleado***, en nuestro caso tenemos ***Gerente*** y ***Oficinista***. Podemos escribir código que funcione con cualquiera de estas clases como instancias de ***IEmpleado***, sin conocer ningún detalle de la clase que implementa ***IEmpleado***.

Por ejemplo, podemos crear un método que envíe a la consola el nombre de un empleado de la siguiente manera.

```
void ImprimeDatos(IEmpleado empleado)
{
    Console.WriteLine($"{empleado.Nombre} {empleado.Apellidos}");
}
```

Podemos consumir este método, proporcionándole un objeto de tipo ***IEmpleado***, ***Gerente*** u ***Oficinista*** como se muestra en el siguiente ejemplo.

```
Gerente Luis = new Gerente();
ImprimeDatos(Luis);
IEmpleado Pedro = new Oficinista();
ImprimeDatos(Pedro);
Oficinista Julia = new Oficinista();
ImprimeDatos(Julia);
```

Si tuviéramos que crear una nueva clase que implemente ***IEmpleado***, una instancia de esta nueva clase también podría ser enviada como argumento del método ***ImprimeDatos***. Con esto, estamos ahorrando la escritura de un método ***ImprimeDatos*** que reciba un tipo ***Gerente***, uno que reciba un tipo ***Oficinista*** y cuando haya nuevas clases, la creación de sus métodos correspondientes.

Implementando múltiples Interfaces

En muchos casos, podríamos querer crear clases que implementan más de una interface. Por ejemplo, podríamos querer:

- Implementar la interface **IDisposable** para permitir al CLR liberar los recursos de nuestra clase correctamente.
- Implementar la interface **IComparable** para permitir a las clases colección ordenar instancias de nuestra clase.
- Implementar nuestra propia Interface personalizada para definir la funcionalidad de nuestra clase.

Supongamos que tenemos empleados que pueden ser sindicalizados, si son sindicalizados deben tener una cuota sindical y alguna otra información adicional. Podríamos crear una Interface donde podamos definir la información adicional que debe almacenarse para un empleado sindicalizado. La interface sería similar a la siguiente.

```
interface ISindicalizado
{
    decimal CuotaSindical { get; set; }
    decimal SueldoDiario { get; }
}
```

Si ahora deseamos crear una clase ***ObreroSindicalizado*** para representar la información de un empleado obrero que se encuentre sindicalizado, podríamos implementar en la clase ***ObreroSindicalizado***, la Interface ***IEmpleado*** y la Interface ***ISindicalizado*** al mismo tiempo.

Para implementar múltiples interfaces, simplemente agregamos a la declaración de la clase, una lista de las interfaces separada por comas. La clase debe implementar cada miembro de cada Interface que agregamos a la declaración de la clase.

El siguiente ejemplo, muestra la declaración de la clase ***ObreroSindicalizado*** indicando que implementa las interfaces ***IEmpleado*** e ***ISindicalizado***.

```
class ObreroSindicalizado : IEmpleado, ISindicalizado
{
}
```

Implementación Implícita y Explícita

Cuando creamos una clase que implementa una interface, debemos elegir si se debe implementar la interface implícita o explícitamente.

Para implementar una interface de forma implícita, debemos implementar cada uno de los miembros de la Interface con una firma que coincida con la definición de los miembros de la interface.

Para implementar una interface de forma explícita, debemos calificar completamente cada nombre de los miembros de tal forma que sea claro que el miembro pertenece a una interface particular.

En el siguiente ejemplo, no sabemos si el miembro ***SueldoDiario*** es la implementación de la Interface ***IEmpleado*** o de la Interface ***ISindicalizado***.

```
class ObreroSindicalizado : IEmpleado, ISindicalizado
{
    public bool this[byte index]
    {
          // ...
    }

    public string Apellidos
    {
           // ...
    }

    public decimal CuotaSindical
    {
           // ...
    }

    public string Nombre
    {
           // ...
    }

    public decimal SueldoDiario
    {
           // ...
    }

    public event EventHandler AlDetectarAusenciaDePago;

    public decimal CalcularPagoMensual(byte diasTrabajados)
    {
           // ...
    }
}
```

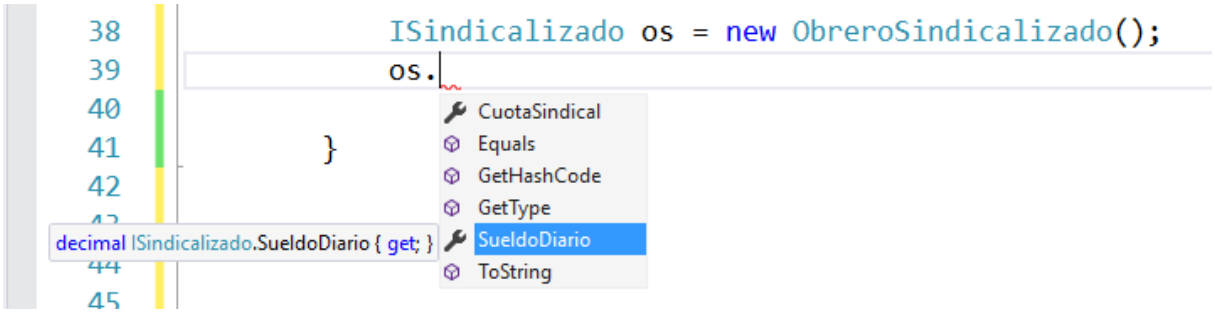
Podemos implementar explícitamente el miembro ***SueldoDiario*** de la Interface ***ISindicalizado*** de la siguiente manera.

```
    decimal ISindicalizado.SueldoDiario
    {
        get
        {
            return 0;
        }
    }
```

En la mayoría de los casos, implementar una interfaz implícitamente o explícitamente es una cuestión de estética. Esto no hace una diferencia en cómo se compila la clase. Algunos desarrolladores prefieren implementar interfaces explicitas porque hacerlo así puede hacer que el código sea más fácil de entender. Un único escenario en el que se debe utilizar implementación de interfaces explicitas es cuando se implementan dos interfaces que comparten un nombre de miembro como en nuestro caso la propiedad ***SueldoDiario***, o bien cuando haya conflictos de nombres entre la clase y la interface. En nuestro ejemplo, el compilador sería incapaz de resolver la referencia a ***SueldoDiario*** sin una implementación explícita.

Cuando implementamos un miembro explícitamente, este ya no es expuesto a través de la clase que implementa. Si declaramos una variable de tipo ***ObreroSindicalizado***, podremos ver que solo se muestra el miembro ***SueldoDiario*** implementado implícitamente pero no aparece el miembro ***SueldoDiario*** implementado explícitamente.

Para poder acceder al miembro implementado explícitamente, debemos accederlo a través de una instancia de un tipo de la Interface ***ISindicalizado***.



Implementando la Interface IComparable

El .NET Framework incluye varias clases colección que permiten ordenar el contenido de una colección. Estas clases, como la clase ***ArrayList***, incluyen un método llamado ***Sort***. Cuando se invoca a este método en una instancia de ***ArrayList***, la colección ordena su contenido.

¿Cómo sabe la instancia ***ArrayList*** la forma en que debe ordenar los elementos de la colección? En el caso de tipos simples, tales como los enteros, esto parece bastante sencillo. Intuitivamente, el 2 le sigue al 1, el 3 le sigue al 2 y así sucesivamente. Sin embargo, supongamos que creamos una colección de objetos ***Empleado***. ¿Cómo determinará una instancia ***ArrayList*** si un ***Empleado*** es mayor o menor que otro ***Empleado***? La instancia ***Arrarylist*** simplemente no podrá ordenar los elementos.

La solución a este problema es que la clase ***Empleado*** debe proporcionar a la instancia ***ArrayList***, la lógica que le permita comparar un ***Empleado*** con otro ***Empleado***. Para hacer esto, la clase ***Empleado*** debe implementar la Interface ***IComparable***.

El siguiente ejemplo, muestra la interface **IComparable**.

```
public interface IComparable
{
    int CompareTo(Object obj);
}
```

Podemos notar que la interface ***IComparable*** declara un único método llamado ***CompareTo***. La implementación de ese método debe:

- Comparar la instancia del objeto actual con otro objeto del mismo tipo (el parámetro).
- Devolver un valor entero que indique si la instancia del objeto actual debería ser colocado antes, en la misma posición o después de la instancia del objeto pasado como parámetro.

El valor entero devuelto por el método ***CompareTo*** es interpretado de la siguiente manera:

- Un valor negativo indica que la instancia del objeto actual es menor que la instancia recibida como parámetro.
- Un valor cero indica que la instancia actual del objeto es igual a la instancia recibida como parámetro.
- Un valor positivo indica que la instancia del objeto actual es mayor que la instancia recibida como parámetro.

En el caso de nuestro ejemplo, de manera intuitiva podríamos pensar que para determinar si un empleado es mayor que otro, necesitaríamos conocer su fecha de nacimiento, sin embargo, nuestra lógica puede ser distinta y tomar como base la fecha de ingreso o el sueldo diario o cualquier campo

o combinación de campos para determinar el criterio de comparación. Todo depende de nuestras necesidades.

Supongamos que decidimos ordenar a los empleados por su fecha de ingreso. En el método ***CompareTo*** debemos comparar la instancia del objeto actual – donde se está ejecutando el código - con otro objeto del mimo tipo - el parámetro -.

El siguiente ejemplo, muestra la clase ***Empleado*** implementando la interface ***IComparable***.

```
class Empleado : IComparable
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public decimal SueldoDiario { get; set; }
    public DateTime FechaDeIngreso { get; set; }

    public int CompareTo(object obj)
    {
        var e = obj as Empleado;
        int Resultado = 0;
        if (e != null)
        {
            if (this.FechaDeIngreso < e.FechaDeIngreso)
            {
                Resultado = -1;
            }
            else if (this.FechaDeIngreso == e.FechaDeIngreso)
            {
                Resultado = 0;
            }
            else
            {
                // this.FechaDeIngreso > e.FechaDeIngreso
                Resultado = 1;
            }
        }
        else
        {
            throw (new Exception("Tipos incompatibles"));
        }
        return Resultado;

    }

}
```

Cuando se invoca al método ***Sort*** en una instancia ***ArrayList***, el método ***Sort*** invoca al método ***CompareTo*** de los miembros de la colección para determinar el orden correcto para la colección.

Implementando la Interface IComparer

Cuando invocamos al método ***Sort*** en una instancia ***ArrayList***, el ***ArrayList*** ordena la colección basándose en la lógica implementada de la Interface ***IComparable*** del tipo que se está ordenando. El creador de la instancia de ***ArrayList*** no tiene ningún control sobre los criterios que se utilizan para ordenar la colección.

En algunos casos, los desarrolladores pueden querer ordenar las instancias de la clase utilizando criterios de ordenación alternativos. Por ejemplo, supongamos que deseamos ordenar una colección de instancias ***Empleado*** por el valor de la propiedad ***SueldoDiario*** en lugar de la propiedad ***FechaDeIngreso***.

Para ordenar una instancia de ***ArrayList*** utilizando criterios de ordenación personalizados, debemos hacer dos cosas:

1. Crear una clase que implemente la interface ***IComparer*** para proporcionar la funcionalidad de ordenación personalizada.
2. Invocar al método ***Sort*** de la instancia de ***ArrayList*** y pasar una instancia de la implementación ***IComparer*** como parámetro.

La interface ***IComparer*** declara un método único llamado ***Compare***. La implementación de este método debe comparar dos objetos del mismo tipo y devolver un valor entero que indique si la instancia del primer parámetro es menor, igual o mayor a la instancia del segundo parámetro. Un proceso similar al seguido para implementar la interface ***IComparable***.

Siguiendo las buenas prácticas de utilizar los métodos de comparación de los tipos integrados (en caso de que exista uno) en lugar de crear uno propio, la clase que implementa la interface ***IComparer*** puede quedar de la siguiente forma.

```
class OrdenIngreso : IComparer
{
    public int Compare(object x, object y)
    {
        Empleado e1 = x as Empleado;
        Empleado e2 = y as Empleado;
        return e1.SueldoDiario.CompareTo(e2.SueldoDiario);
    }
}
```

El siguiente ejemplo, muestra cómo utilizar la implementación de la interface ***IComparer***.

```
        ArrayList Empleados = new ArrayList();
        // Agregar elementos a la colección
        // ....

        // Ordenar la colección
        Empleados.Sort(new OrdenIngreso());
```

Lección 3:
Implementando colecciones de tipos seguros (Type-safe Collections)

Casi todas las aplicaciones que creamos, de alguna u otra manera, utilizan clases Collection. En la mayoría de los casos, contienen un conjunto de objetos del mismo tipo. Cuando interactuamos con una colección, frecuentemente deseamos que la colección contenga objetos del mismo tipo. Históricamente, esto ha creado diversos desafíos. Hemos tenido que crear lógica de manejo de excepciones en caso de que una colección contenga elementos de tipos incorrectos. También hemos tenido que empaquetar (*Boxing*) tipos valor para poder agregarlos a colecciones y desempaquetarlos (*Unboxing*) para poder recuperarlos. Visual C# elimina muchos de esos desafíos mediante el uso de tipos G*enéricos*.

En esta lección, aprenderemos a crear y utilizar clases *Genéricas* para crear colecciones de tipos seguros de cualquier tipo.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Describir los tipos *Genéricos*.
- Identificar las ventajas de las clases *Genéricas* sobre las clases *No Genéricas*.
- Aplicar restricciones a los tipos *Genéricos*.
- Utilizar colecciones *List Genéricas*.
- Utilizar colecciones *Dictionary Genéricas*.
- Crear colecciones *Genéricas* personalizadas.
- Crear colecciones *Genéricas enumerables*.

Introducción a los Tipos Genéricos

La característica ***Generics*** o en español ***Genéricos***, permite crear y utilizar colecciones fuertemente tipadas que son de tipo seguro, no requiere que realicemos un *cast* de los elementos y no requiere que hagamos box y unbox de los tipos valor. Las clases genéricas funcionan mediante la inclusión de un parámetro para indicar un Tipo (tradicionalmente representado como **T**), después del nombre en la declaración de la clase o Interface. No necesitamos especificar el tipo de **T** hasta que creamos una instancia de la clase. Para crear una clase genérica, necesitamos hacer lo siguiente:

- Agregar el parámetro de Tipo entre los símbolos *Menor que (<)* y *Mayor que (>)* después del nombre de la clase.
- Utilizar el parámetro de Tipo en lugar del nombre del Tipo en los miembros de la clase.

El siguiente ejemplo muestra cómo crear una clase *Genérica*.

```
class ListaGenerica<T>
{
    public void Add(T item)
    {
        // Lógica para agregar un elemento.
    }
    public void Remove(T item)
    {
        // Lógica para eliminar un elemento.
    }
}
```

Cuando declaramos una variable de nuestro tipo genérico, debemos especificar el tipo que deseamos proporcionar como un parámetro de Tipo. Por ejemplo, si deseamos utilizar la lista personalizada para almacenar objetos de tipo ***Empleado***, debemos proporcionar ***Empleado*** como el parámetro de Tipo.

El siguiente ejemplo muestra como instanciar una clase genérica.

```
ListaGenerica<Empleado> Empleados = new ListaGenerica<Empleado>();
Empleado Luis = new Empleado();
Empleado Maria = new Empleado();
Empleados.Add(Luis);
Empleados.Add(Maria);
```

Al crear una instancia de la clase genérica, cada instancia de ***T*** dentro de la clase, es remplazada con el tipo de parámetro que proporcionamos. Por ejemplo, al instanciar la clase ***ListaGenerica*** con el parámetro de tipo ***Empleado***, los métodos ***Add*** y ***Remove*** solo aceptarán argumentos de tipo ***Empleado***.

Ventajas de los Tipos Genéricos

El uso de clases genéricas, especialmente para las colecciones, ofrece tres ventajas principales sobre las clases no genéricas: Seguridad de Tipos, No Casting, No boxing y unboxing.

Seguridad de tipos (Type Safety)

Consideremos un ejemplo donde utilicemos un ***ArrayList*** para almacenar una colección de objetos de tipo ***int***. Podemos agregar objetos de cualquier tipo a un objeto ***ArrayList***. Supongamos que un desarrollador agrega un objeto de tipo ***String*** a la colección. El código compilará sin problema. Sin embargo, se producirá una excepción en tiempo de ejecución si el método ***Sort*** es invocado debido a que la colección es incapaz de comparar objetos de diferentes tipos. Por otra parte, al recuperar un objeto de la colección, debemos convertir el objeto al tipo correcto. Si intentamos convertir el objeto al tipo equivocado, una excepción de conversión inválida se generará en tiempo de ejecución.

Como una alternativa al uso de ***ArrayList***, supongamos que utilizamos una colección **List<T>** genérica para almacenar la colección de objetos de tipo ***int***. Al crear una instancia de la lista, se proporciona un argumento de tipo ***int***. En este caso, queda garantizado que la lista es homogénea debido a que el código no compilará si se intenta añadir un objeto de cualquier otro tipo distinto a ***int***. El método ***Sort*** trabajará sin problema debido a que su colección es homogénea. Finalmente, el indizador devuelve los objetos de tipo ***int*** en lugar de ***System.Object***, por lo que no hay riesgo de excepciones de conversión no válida.

No Casting

El casting es un proceso computacionalmente caro. Cuando se agregan elementos a un objeto ***ArrayList***, los elementos son convertidos implícitamente al tipo **System.Object**. Al recuperar los elementos del ***ArrayList***, debemos convertirlos explícitamente de regreso a su tipo original. El uso de genéricos para agregar y recuperar elementos sin conversiones, mejora el rendimiento de la aplicación.

No boxing y unboxing

Si deseamos almacenar Tipos Valor en un ***ArrayList***, los elementos deben ser “encajonados” (*Boxing*) cuando se agregan a la colección y “desencajonados” (*Unboxing*) cuando se recuperan. *Boxing* y *Unboxing* incurren en un gran costo computacional y pueden reducir significativamente el rendimiento de las aplicaciones, especialmente cuando iteramos sobre colecciones grandes. En contraste, podemos agregar tipos valor a las listas genéricas sin encajonar (*Boxing*) o desencajonar (*UnBoxing*) el valor.

Restricciones en los Tipos Genéricos

En algunos casos, puede que necesitemos restringir los tipos que los desarrolladores pueden suministrar como argumentos cuando instancien nuestra clase genérica. La naturaleza de estas restricciones dependerá de la lógica que nosotros implementemos en la clase genérica. Por ejemplo, si una clase utiliza una propiedad denominada ***SueldoDiario*** de la siguiente manera:

```
class Operaciones<T>
{
    T Empleado;
    public Operaciones()
    {
        Empleado.SueldoDiario = 0;
    }
}
```

Para que podamos compilar la aplicación, será necesario restringir el parámetro de tipo a las clases que incluyen la propiedad ***SueldoDiario***. Supongamos que la propiedad ***SueldoDiario*** está definida por la Interface ***IEmpleado***, para implementar esta restricción, podríamos limitar el parámetro de tipo a clases que implementen la interface ***IEmpleado*** utilizando la palabra clave ***where*** de la siguiente manera:

```
class Operaciones<T> where T : IEmpleado
{
    T Empleado;
    public Operaciones()
    {
        Empleado.SueldoDiario = 0;
    }
}
```

Podemos aplicar seis tipos de restricciones a los parámetros de tipo.

Restricción	Descripción
where T : <nombre de la interface>	Restricción por Interface. Esta restricción implica que el tipo T debe ser o implementar la interface especificada.
where T : <nombre de la clase base>	Restricción por clase. Esta restricción implica que el tipo T debe ser o derivar de la clase especificada.
where T : U	Restricción por tipos derivados. Esta restricción implica que el tipo T debe ser o derivar del tipo U.
where T : new()	Restricción por constructor. Esta restricción implica que el tipo T debe tener un constructor público predeterminado.
where T : struct	Restricción por tipo valor. Esta restricción implica que el tipo T debe ser un tipo valor.
where T : class	Restricción por tipo referencia. Esta restricción implica que el tipo T debe ser un tipo referencia.

Es posible aplicar restricciones a múltiples parámetros y múltiples restricciones a un simple parámetro como se muestra en el siguiente ejemplo.

```
class Empleado<T, U>
where U : struct
where T : Gerente, new()
{
}
```

En el ejemplo anterior, podemos ver que se están aplicando restricciones a los parámetros ***U*** y ***T***. También podemos ver que se están aplicando 2 restricciones al parámetro ***T.***

Utilizando Colecciones List Genéricas

Uno de los usos más comunes e importantes de los tipos genéricos está en las clases de colecciones. Las colecciones genéricas se dividen en dos grandes categorías: las colecciones genéricas de *Listas* y las colecciones genéricas de *Diccionarios*.

Una *Lista Genérica* almacena una colección de objetos de tipo **T**.

La clase List<T>

La clase ***List<T>*** proporciona una alternativa de tipos seguros a la clase ***ArrayList***. Al igual que la clase ***ArrayList***, la clase ***List<T>*** incluye métodos para:

- Agregar un elemento.
- Eliminar un elemento.
- Insertar un elemento en el índice especificado.
- Ordenar los elementos de la colección utilizando el comparador predeterminado o un comparador especificado.

El siguiente ejemplo, muestra cómo utilizar la clase ***List<T>***.

```
// Crear una instancia de una colección de tipos seguros.
List<int> Enteros = new List<int>();

// Agregar elementos a la colección.
Enteros.Add(3);
Enteros.Add(30);
Enteros.Add(4);
Enteros.Add(40);
Enteros.Add(5);
Enteros.Add(50);

// Ordenar la colección.
Enteros.Sort();

// Eliminar un elemento de la colección.
Enteros.Remove(3);

// Insertar en el índice especificado.
Enteros.Insert(0, 13);

// Recorrer la colección.
foreach (int entero in Enteros)
{
    Console.WriteLine(entero);
}
```

Otras clases genéricas List

El espacio de nombres ***System.Collections.Generic***, también incluye varias colecciones genéricas que proporcionan funcionalidad más especializada:

- La clase ***LinkedList<T>***, proporciona una colección genérica en la que cada elemento está enlazado con el elemento anterior de la colección y con el elemento siguiente de la colección. Cada elemento de la colección está representado por un objeto ***LinkedListNode<T>*** que contiene un valor de tipo ***T***, una referencia a la instancia padre ***LinkedList<T>***, una referencia al elemento anterior de la colección y una referencia al siguiente elemento de la colección.

- La clase ***Queue<T>***, representa una colección de objetos de tipos seguros "*Primero en entrar, primero en salir*".

- La clase ***Stack<T>***, representa una colección de objetos de tipos seguros *"Último en entrar, primero en salir"*.

Siempre que sea posible, en lugar de utilizar clases no genéricas, debemos utilizar clases genéricas.

Utilizando Colecciones Dictionary Genéricas

Las clases ***Diccionario*** almacenan colecciones de parejas llave valor. El valor es el objeto que deseamos almacenar y la llave es el objeto que utilizamos para indexar y recuperar el valor. Por ejemplo, podemos utilizar una clase diccionario para almacenar datos de empleados donde la llave es el ID del empleado y el valor son los datos adicionales del empleado. En el caso de los diccionarios genéricos, tanto la llave como el valor, son de tipos seguros.

La clase Dictionary<TKey, TValue>

La clase ***Dictionary<TKey, TValue>*** proporciona una clase diccionario de tipos seguros de propósito general. Podemos agregar valores duplicados a la colección, pero las llaves deben ser únicas. La clase disparará una excepción de tipo ***ArgumentException*** si intentamos agregar una llave que ya exista en el diccionario.

El siguiente ejemplo, muestra cómo utilizar la clase ***Dictionary<TKey, TValue>***.

```
// Crear una colección de Empleados con int como llaves y Empleado como valores.
Dictionary<int, Empleado> Empleados =
    new Dictionary<int, Empleado>();

// Agregar elementos al diccionario.
// La clase disparará una excepción de tipo ArgumentException si intentamos
// agregar una llave que ya exista en el diccionario.
var Juan = new Empleado(1, "Juan", "Rojas");
var Maria = new Empleado(2, "Maria", "Sanders");
var Luis = new Empleado(3, "Luis", "Rojas");
Empleados.Add(Juan.ID, Juan);
Empleados.Add(Maria.ID, Maria);
Empleados.Add(Luis.ID, Luis);

// Recuperar un valor por su llave.
// Si intentamos extraer un elemento que no existe,
// se generará una excepción KeyNotFoundException.
var MariaSanders = Empleados[2];

// El método TryGetValue permite recuperar un elemento del diccionario.
// Devuelve True si la llave existe y False si no existe.
Empleado E;
if (Empleados.TryGetValue(10, out E))
{
    // El empleado con ID = 10 existe y se encuentra en la variable E.
}
else
{
    // El empleado con ID = 10 no existe.
}

// Podemos utilizar su indexador para modificar el valor asociado con la llave.
Empleados[2] = new Empleado(2, "Pedro", "Aguilar");
```

Otras clases genéricas diccionario

Las clases ***SortedList<TKey, TValue>*** y ***SortedDictionary<TKey, TValue>*** proporcionan diccionarios genéricos en los cuales los elementos son ordenados por sus llaves según la implementación de la interface ***IComparer<T>*** asociada.

SortedList y ***SortedDictionary*** son similares, su diferencia radica en el uso de memoria y en la velocidad de inserción y eliminación:

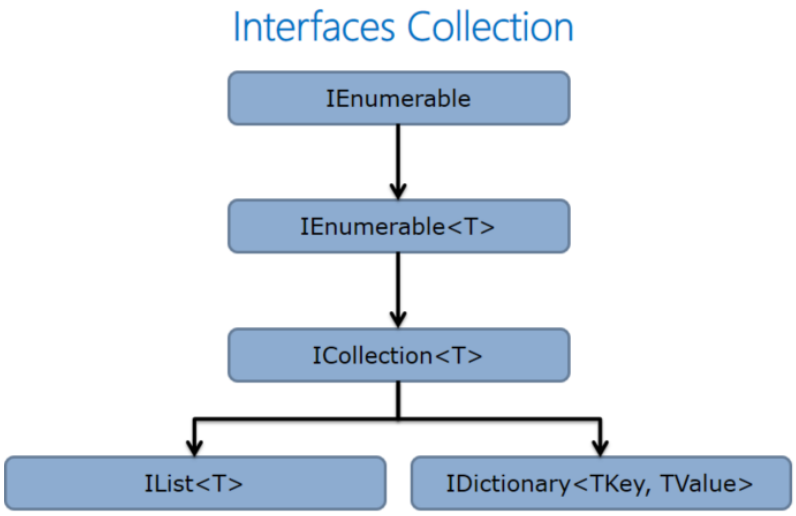
- ***SortedList*** utiliza menos memoria que ***SortedDictionary***.
- Las operaciones de inserción y eliminación de ***SortedDictionary*** con datos no ordenados, son más rápidas en comparación de ***SortedList***.
- Si la lista se llena de una sola vez con datos ordenados, la colección ***SortedList*** es más rápida que ***SortedDictionary***.

Utilizando Interfaces Collection

El espacio de nombres ***System.Collections.Generic*** proporciona diversas colecciones genéricas para adaptarse a diferentes escenarios. Sin embargo, habrá circunstancias donde necesitemos crear nuestras propias clases de colecciones genéricas con el fin de proporcionar funcionalidad más especializada. Por ejemplo, podríamos necesitar almacenar datos en una estructura de árbol o crear una lista ligada circular.

¿Por dónde empezar cuando necesitemos crear una clase colección personalizada? Todas las colecciones tienen ciertas cosas en común. Por ejemplo, normalmente queremos ser capaces de enumerar los elementos de la colección mediante un ciclo ***foreach*** y necesitamos métodos para agregar elementos, eliminar elementos y borrar toda la lista.

Como se muestra en la siguiente imagen, el .NET Framework proporciona un conjunto jerárquico de interfaces que definen las características y comportamientos de las colecciones. Estas interfaces se construyen una sobre otra para definir progresivamente una funcionalidad más específica.



Las interfaces IEnumerable e IEnumerable<T>

Si queremos ser capaces de utilizar un ciclo **foreach** para enumerar los elementos de una colección genérica personalizada, debemos implementar la interfaz ***IEnumerable<T>***. La Interface ***IEnumerable<T>*** define un método único llamado ***GetEnumerator()***. Este método debe devolver un objeto de tipo ***IEnumerator<T>***. La sentencia **foreach** se basa en este objeto enumerador para iterar a través de la colección.

La Interface ***IEnumerable<T>***, extiende la definición de la Interface ***IEnumerable*** que también define un método llamado ***GetEnumerator()***. Cuando una Interface extiende otra interface, esta expone

todos los miembros de la interface padre. En otras palabras, si implementamos ***IEnumerable<T>***, también necesitamos implementar la interface ***IEnumerable***.

La interface ICollection

La interface ***ICollection<T>***, define la funcionalidad básica que es común a todas las colecciones genéricas. La interface extiende a la interface ***IEnumerable<T>***, lo que significa que si queremos implementar ***ICollection<T>***, también debemos implementar los miembros definidos por la interface ***IEnumerable<T>*** e ***IEnumerable***.

La interface ***ICollection<T>*** define los siguiente métodos:

Nombre	Descripción
Add	Agrega un elemento de tipo **T** a la colección.
Clear	Elimina todos los elementos de la colección.
Contains	Indica si la colección contiene un valor específico.
CopyTo	Copia los elementos de la colección a un objeto **Array** a partir de un índice determinado de **Array**.
Remove	Elimina un elemento específico de la colección.
GetEnumerator	Devuelve un enumerador que permite iterar sobre la colección

La interface ***ICollection<T>*** define las siguientes propiedades:

Nombre	Descripción
Count	Obtiene el número de elementos de la colección.
IsReadOnly	Indica si la colección es de solo lectura.

La interface IList<T>

La interface ***IList<T>,*** define la funcionalidad básica de las clases de listas genéricas. Debemos implementar esta interface si deseamos definir una colección lineal de los valores. Además de los miembros definidos en la Interface ***ICollection<T>***, la interface ***IList<T>*** define métodos y propiedades que permiten utilizar índizadores para trabajar con los elementos de la colección. Por ejemplo, si creamos una lista llamada *MiLista*, podemos utilizar *MiLista[0]* para acceder al primer elemento de la colección.

La interface ***IList<T>*** define los siguiente métodos:

Nombre	Descripción
IndexOf	Determina el índice de un elemento específico de la lista.
Insert	Inserta un elemento en la colección en el índice especificado.
RemoveAt	Elimina el elemento que se encuentra en el índice especificado de la colección.

La interface ***IList<T>*** define la siguiente propiedad:

Nombre	Descripción
Item	Obtiene o establece el elemento que se encuentra en el índice especificado.

La interface IDictionary<TKey, TValue>

La interface ***IDictionary<TKey, TValue>*** define la funcionalidad básica para las clases genéricas diccionario. Se debe implementar esta interface si estamos definiendo una colección de parejas llave-valor. Además de los miembros definidos en la interface ***ICollection<T>***, la interface ***IDictionary<T>*** define métodos y propiedades que son específicas para trabajar con parejas llave-valor.

La interface ***IDictionary<TKey, TValue>*** define los siguiente métodos:

Nombre	Descripción
Add(TKey, TValue)	Agrega un elemento a la colección con la llave y el valor especificados.
ContainsKey	Indica si la colección incluye un elemento con la llave especificada.
Remove(TKey)	Elimina el elemento de la colección con la llave especificada.
TryGetValue	Intenta establecer el valor de un parámetro de salida con el valor asociado a la llave especificada. Si existe la llave, el método devuelve ***true***. Si la llave no existe, el método devuelve ***false*** y el parámetro de salida no es modificado.

La interface ***IDictionary<TKey, TValue>*** define las siguientes propiedades:

Nombre	Descripción
Item	Obtiene o establece el valor de un elemento de la colección basado en una llave especificada. Esta propiedad permite utilizar la notación de indizador, por ejemplo, ***MiDiccionario[MiLLave] = MiValor***.
Keys	Devuelve las llaves de la colección como una instancia ***ICollection<T>***.
Values	Devuelve los valores de la colección como una instancia ***ICollection<T>***.

Para obtener información completa y ejemplos de todas las interfaces genéricas cubiertas en este tema, se recomienda visitar el siguiente enlace:

System.Collections.Generic Namespace
http://go.microsoft.com/fwlink/?LinkID=267802

Implementando la Interface IEnumerable

Para enumerar una colección, normalmente utilizamos un ciclo ***foreach***. El ciclo ***foreach*** expone cada elemento de la colección, uno a la vez, en un orden apropiado para la colección. La sentencia ***foreach*** oculta algunas de las complejidades de las enumeraciones. Para que la sentencia ***foreach*** trabaje, una clase colección debe implementar la interface ***IEnumerable***. Esta interface expone un método, ***GetEnumerator***, que debe devolver un tipo ***IEnumerator***.

El siguiente ejemplo muestra una clase genérica implementando la interface ***IEnumerable***.

```
class Cuadrado<T>:IEnumerable<T>
{
    // Propiedades de la clase Cuadrado que definen los vértices de un cuadrado.
    public T V1 { get; set; }
    public T V2 { get; set; }
    public T V3 { get; set; }
    public T V4 { get; set; }

    // Miembros de la interface IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        throw new NotImplementedException();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Al implementar la interface ***IEnumerable<T>*** en una clase genérica, podemos notar que la interface define el método ***GetEnumerator*** que devuelve un tipo ***IEnumerator<T>*** y un método privado ***GetEnumerator*** que devuelve un tipo ***IEnumerator*** no genérico implementado explícitamente. Este miembro es definido por la Interface no genérica ***IEnumerable*** que es extendida por la interface ***IEnumerable<T>*** genérica.

Debido a que la clase de nuestro ejemplo es una clase genérica, en el método ***GetEnumerator*** no genérico, podemos devolver simplemente el enumerador genérico de la siguiente manera:

```
IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}
```

Después de esto, necesitamos crear el enumerador que debemos devolver en el método genérico ***GetEnumerator***.

Implementando la Interface IEnumerator

La interface ***IEnumerator<T>***, define la funcionalidad que todos los enumeradores deben implementar. Es una práctica común anidar la clase enumerador dentro de la clase que expone la colección como se muestra en el siguiente ejemplo.

```
class Cuadrado<T>:IEnumerable<T>
{
    // Propiedades de la clase Cuadrado que definen los vértices de un cuadrado.
    public T V1 { get; set; }
    public T V2 { get; set; }
    public T V3 { get; set; }
    public T V4 { get; set; }

    // Miembros de la interface IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        throw new NotImplementedException();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }

    // Enumerador
    class Enumerador : IEnumerator<T>
    {
    }

}
```

La clase enumerador debe ser capaz de enumerar los elementos de la colección. Podemos proporcionarle la instancia donde se encuentran los elementos a enumerar a través del constructor como se muestra en el siguiente ejemplo.

```
class Enumerador : IEnumerator<T>
{
    Cuadrado<T> Instancia;
    int Index = -1;
    public Enumerador(Cuadrado<T> instanciaEnumerable)
    {
        Instancia = instanciaEnumerable;
    }
}
```

Un enumerador es esencialmente un apuntador a los elementos de la colección. Inicialmente el apuntador se encuentra antes del primer elemento.

La interface ***IEnumerator*** define la propiedad ***Current*** que, durante la enumeración, devuelve el elemento al que el enumerador esté apuntando actualmente.

En nuestro ejemplo, auxiliándonos de la variable ***Index***, podemos implementar la propiedad ***Current*** de la siguiente manera.

```
public T Current
{
    get
    {
        T VerticeActual;
        switch (Index)
        {
            case 0:
                VerticeActual = Instancia.V1;
                break;
            case 1:
                VerticeActual = Instancia.V2;
                break;
            case 2:
                VerticeActual = Instancia.V3;
                break;
            case 3:
                VerticeActual = Instancia.V4;
                break;
            default:
              // La palabra clave default devolverá null si T es un tipo
              // referencia y un valor cero para valores numéricos. Para las
              // estructuras, devolverá la estructura con sus miembros inicializados
              // con null o ceros dependiendo si son tipo referencia o valor.
                VerticeActual = default(T);
                break;
        }
        return VerticeActual;
    }
}
```

La interface ***IEnumerator<T>*** define también el método ***Dispose*** que permite liberar recursos no administrados. En nuestro ejemplo, no manejamos recursos no administrados así que podemos dejarlo sin código.

```
public void Dispose() { }
```

La interface ***IEnumerator*** también expone la propiedad privada ***Current*** de la interface no genérica ***IEnumerator*** que extiende. Aquí podemos devolver el valor de la propiedad ***Current*** de tipo genérico.

```
object IEnumerator.Current
{
    get { return this.Current; }
}
```

Cuando se invoca al método ***MoveNext*** definido por la interface ***IEnumerator***, el puntero avanza al siguiente elemento de la colección y devuelve **true** si el enumerador fue capaz de avanzar una posición, o **false** si se ha llegado al final de la colección. De esta forma podemos indicar el final de la colección.

```
public bool MoveNext()
{
    bool Resultado;
    if (Index < 3)
    {
        Index++;
        Resultado = true;
    }
    else
    {
        Resultado = false;
    }
    return Resultado;
}
```

Finalmente, el método ***Reset*** definido por la interface ***IEnumerator*** establece el enumerador en su posición inicial, que es antes del primer elemento de la colección.

```
public void Reset()
{
    Index = -1;
}
```

Cuando se crea un enumerador, se debe definir:

- Qué elemento del enumerador debe considerarse como el primer elemento de la colección.
- En qué orden el enumerador debe moverse a través de los elementos de la colección.

Como vimos previamente, la interface ***IEnumerable<T>*** define un método llamado ***GetEnumerator*** que devuelve el enumerador de la clase colección como una instancia ***IEnumerator<T>***. Este es el enumerador que un ciclo ***foreach*** utilizará.

El siguiente ejemplo, muestra el código del método ***GetEnumerator*** de la clase ***Cuadrado***.

```
public IEnumerator<T> GetEnumerator()
{
    return new Enumerador(this);
}
```

Implementando IEnumerator a través de un Iterador

Como hemos visto, podemos proporcionar un enumerador mediante la creación de una clase personalizada que implemente la interface ***IEnumerator<T>***. Sin embargo, si la clase colección personalizada utiliza un tipo ***IEnumerable*** para almacenar datos o bien si tenemos elementos finitos que podamos enumerar fácilmente, podemos utilizar un ***iterador*** para implementar la interface ***IEnumerable<T>*** sin tener que proporcionar una implementación de la interface ***IEnumerator<T>***.

El siguiente ejemplo muestra la forma de utilizar iteradores en una clase para implementar un enumerador.

```
public IEnumerator<T> GetEnumerator()
{
    yield return V1;
    yield return V2;
    yield return V3;
    yield return V4;
}
```

Al utilizar la palabra clave ***yield*** estamos indicando que el método en el que aparece es un iterador. Lo único que tenemos que hacer es retornar cada uno de los valores de la colección.

Si los elementos estuvieran en una colección, podríamos utilizar un ciclo ***foreach*** para devolver cada valor con una sentencia ***yield return*** como se muestra en el siguiente ejemplo.

```
private List<T> Vertices = new List<T>();

// ...

public IEnumerator<T> GetEnumerator()
{
    foreach (var vertice in Vertices)
    {
        yield return vertice;
    }
}
```

Cuando se invoca al método ***GetEnumerator***, se utiliza una sentencia ***yield return*** que define al iterador para devolver cada elemento de la colección. Esencialmente, la sentencia ***yield return*** detiene la ejecución para devolver el elemento actual al invocador antes de que el siguiente elemento de la secuencia sea recuperado. De esta manera, aunque el método ***GetEnumerator*** no parece devolver un tipo ***IEnumerator***, el compilador es capaz de construir un enumerador a partir de la lógica de iteración que nosotros proporcionamos.

Introducción a C#

Módulo 5: Creando una jerarquía de Clases utilizando Herencia

Acerca del módulo

El concepto de herencia es fundamental para la programación orientada a objetos en cualquier lenguaje. También es una de las herramientas más poderosas de nuestra caja de herramientas de desarrollador. Esencialmente, la herencia nos permite crear nuevas clases al heredar características y comportamientos de clases existentes. Al heredar de una clase existente y agregar nuestra propia funcionalidad, nuestra clase se convierte en una instancia más especializada de la clase existente. Esto no sólo nos ahorra tiempo al reducir la cantidad de código que necesitamos escribir, sino que también nos permite crear jerarquías de clases relacionadas que después se pueden utilizar indistintamente dependiendo de nuestras necesidades.

En este módulo, aprenderemos a utilizar herencia para crear jerarquías de clase y extender los tipos del .NET Framework.

Objetivos

Al finalizar este módulo, los participantes contarán con las habilidades y conocimientos para:

- Crear clases Base.
- Crear clases Derivadas mediante el uso de herencia.
- Crear clases que hereden de clases del .NET Framework.

Los temas que se cubren en este módulo son:

- Lección 1: Creando Jerarquías de Clases.
- Lección 2: Extendiendo Clases del .NET Framework.

Lección 1:
Creando Jerarquías de Clases

En lugar de crear nuevas clases desde cero, en la mayoría de los casos podemos utilizar clases existentes como una base para las nuevas clases. Esto es conocido como herencia. La clase hereda los miembros de la clase base y nosotros simplemente incluimos la funcionalidad que queremos agregar a las capacidades de la clase base. De esta forma, nuestra clase se convierte en una versión más especializada de la clase base. El concepto de herencia es uno de los principales pilares de la Programación Orientada a Objetos.

En esta lección, aprenderemos a utilizar herencia para crear jerarquías de clases.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con los conocimientos y habilidades para:

- Describir el concepto de Herencia.
- Crear clases Base.
- Crear miembros de clases Base.
- Crear clases que hereden de clases Base.
- Invocar desde la clase *Derivada*, los *Métodos* y *Constructores* de la clase *Base*.

¿Qué es Herencia?

La Herencia junto con la encapsulación y el polimorfismo, es uno de los 3 pilares fundamentales de la Programación Orientada a Objetos.

En lugar de crear nuevas clases a partir de cero, en la mayoría de los casos podemos utilizar una clase existente como base para la nueva clase. Esto es conocido como Herencia: "*Utilizar una clase existente como base para una nueva clase*".

La herencia permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento definido en otras clases.

La clase hereda todos los *Campos*, *Propiedades*, *Métodos*, *Eventos* e *Indizadores* no privados de la clase base, y nosotros simplemente incluimos la funcionalidad que deseamos agregar a las capacidades de la clase base. De esta manera, la clase se convierte en una versión más especializada de la clase base. En Visual C#, una clase puede heredar de otra clase.

Cuando creamos una clase que hereda de otra clase, la nueva clase es conocida como ***Clase Derivada*** y la clase que de la cual se hereda se conoce como la ***Clase Base***.

Aunque la clase derivada hereda todos los Campos, Propiedades, Métodos, Eventos e Indizadores no privados de la clase base, los *Constructores* y *Destructores* de la clase Base no se heredan.

En la programación orientada a objetos, los términos ***Deriva*** y ***Hereda*** se utilizan indistintamente. Por ejemplo, decir que la clase ***Empleado*** deriva de la clase ***Persona*** significa lo mismo que decir que la clase ***Empleado*** hereda de la clase ***Persona***.

Por ejemplo, en lugar de crear una clase desde cero para representar un ***Empleado*** que es una ***Persona***, podemos crear una clase ***Empleado*** que herede de la clase ***Persona***. La clase derivada ***Empleado*** hereda los miembros de la clase ***Persona***. Dentro de la clase derivada ***Empleado***, sólo tenemos que añadir los miembros que son específicos de un ***Empleado***.

La herencia permite crear jerarquías de clases cada vez más especializadas. Por ejemplo, en lugar de crear una clase ***Obrero*** desde cero, podemos heredar de una clase base más general ***Empleado*** para proporcionar un punto de partida para nuestra funcionalidad. La herencia también puede ayudar a simplificar el mantenimiento del código.

Una clase derivada puede tener únicamente una clase base directa, sin embargo, la herencia es transitiva. Si la clase ***Obrero*** es derivada de la clase ***Empleado*** y la clase ***Empleado*** es derivada de la clase ***Persona***, la clase ***Obrero*** hereda los miembros declarados en la clase ***Empleado*** y en la clase ***Persona***.

Vale la pena mencionar que los tipos ***Struct*** no soportan herencia, pero si pueden implementar Interfaces.

Para obtener más información acerca de Herencia en C#, se recomienda visitar el siguiente enlace:

Inheritance (C# Programming Guide)
http://msdn.microsoft.com/en-us/library/ms173149.aspx

Creando Clases Base: Clases *Abstractas* y Clases *Selladas*

Cuando creamos una clase, debemos considerar si nosotros u otros desarrolladores necesitaremos utilizarla como base para Clases derivadas. Tenemos el control completo para decidir si la clase se puede heredar y como puede ser heredada.

Clases y Miembros Abstractos

Como parte de un diseño orientado a objetos, es posible que deseemos crear clases que sirvan únicamente como clases base para otros tipos. La clase base puede contener funcionalidad faltante o incompleta o incluso podríamos desear que la clase no pueda ser instanciada. En estos casos, podemos agregar la palabra clave ***abstract*** a la declaración de la clase.

El siguiente ejemplo, muestra como declarar una clase abstracta.

```
abstract class Persona
{
}
```

La palabra clave ***abstract*** indica que la clase no puede ser instanciada directamente, sino que existe solamente para definir o implementar miembros de las clases derivadas. Si intentamos crear una instancia de una clase abstracta, obtendremos un error al generar el código.

El propósito de una clase abstracta es proporcionar una definición común de una clase base que múltiples clases derivadas puedan compartir. Por ejemplo, una biblioteca de clases puede definir una clase abstracta que sea utilizada como parámetro para muchos de sus métodos y solicitar a los programadores que utilicen esa biblioteca para proporcionar su propia implementación de la clase mediante la creación de una clase derivada.

Una clase abstracta puede contener miembros abstractos y miembros no abstractos. Los miembros abstractos también se declaran con la palabra clave ***abstract*** y son conceptualmente similares a los miembros de una Interface desde el punto de vista en que los miembros abstractos definen la firma del miembro, pero no proporcionan la implementación. Cualquier clase que herede de la clase abstracta, debe proporcionar una implementación para los miembros abstractos. Los miembros no abstractos, sin embargo, pueden ser utilizados directamente por las clases derivadas.

El siguiente ejemplo, ilustra la diferencia entre miembros abstractos y no abstractos.

```
abstract class Persona
{
    // Una clase abstracta puede tener miembros no abstractos.
    // Las clases derivadas pueden utilizar estos miembros sin modificarlos.
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
```

```
        // Una clase abstracta puede tener miembros abstractos.
        // Las clases derivadas deben sobrescribir e implementar estos miembros.
        abstract public string GenerarID();
    }
```

Únicamente se puede incluir a miembros abstractos dentro de clases abstractas. Una clase no abstracta, no puede incluir a miembros abstractos.

Clases Selladas (sealed)

Es posible también que queramos crear clases que no puedan ser heredadas. Podemos evitar que los desarrolladores hereden de nuestra clase, marcando la clase con la palabra clave ***sealed***.

El siguiente ejemplo, muestra cómo podemos utilizar el modificador ***sealed***.

```
sealed public class Animal
{
}
```

Podemos aplicar el modificador ***sealed*** a las clases que heredan de otras clases y a las clases que implementan interfaces. Una clase sellada no se puede utilizar como clase base. Por esta razón, tampoco puede ser una clase abstracta. Las clases selladas evitan la derivación mientras que una clase abstracta debe ser heredada, ambas se contraponen. Puesto que nunca se pueden utilizar como una clase base, algunas optimizaciones en tiempo de ejecución pueden hacer que sea un poco más rápido invocar a miembros de clases selladas.

Cualquier clase estática es también una clase sellada. No se puede heredar de una clase estática. Del mismo modo, los miembros estáticos dentro de clases no estáticas, no son heredadas por las clases derivadas.

Creando miembros de la Clase Base

Es posible que deseemos implementar un método en una clase base y permitir que las clases derivadas reemplacen la implementación del método con una funcionalidad más específica. Para crear un miembro que los desarrolladores puedan reemplazar en una clase derivada, utilizamos la palabra clave ***virtual***.

El siguiente ejemplo, muestra cómo crear un método virtual en una clase.

```
public class Usuario
{
    public virtual string CalcularID()
    {
        // Esta es la implementación predeterminada del método CalcularID.
        // Debido a que el método es declarado virtual, podemos sobrescribir la
        // implementación en las clases derivadas.
        return System.Guid.NewGuid().ToString();
    }
}
```

Cuando creamos una clase, podemos utilizar modificadores de acceso para controlar la accesibilidad de los miembros de nuestra clase en los tipos derivados. Podemos utilizar los siguientes modificadores de acceso para los miembros de la clase:

Modificador de acceso	Descripción
public	El miembro está disponible para el código que se ejecuta en cualquier Assembly.
protected	El miembro está disponible únicamente dentro de la clase contenedora, o en clases derivadas de la clase contenedora.
internal	El miembro está disponible únicamente en el código dentro del Assembly actual pero no desde otro Assembly.
protected internal	El miembro está disponible para cualquier código dentro del Assembly actual, y para los tipos derivados de la clase contenedora en cualquier Assembly.
private	El miembro está disponible únicamente dentro de la clase contenedora.

Los miembros de una clase son privados de manera predeterminada. Los miembros privados no son heredados por las clases derivadas. Si queremos hacer disponible un miembro privado a la clase derivada, utilizamos explícitamente el modificador de acceso ***protected*** sin el modificador ***private***.

Heredando desde la Clase Base

Para heredar de otra clase, en la declaración de la Clase derivada, debemos agregar dos puntos y a continuación, el nombre de la clase base.

El siguiente ejemplo, muestra como heredar de la clase base.

```
public class Administrador : Usuario
{
}
```

La clase derivada hereda los miembros de la clase base. Dentro de la clase derivada, podemos agregar nuevos miembros para ampliar la funcionalidad del tipo base. Una clase sólo puede heredar de una clase base. Sin embargo, la clase puede implementar una o más interfaces además de derivar de un tipo base.

Sobrescribiendo miembros de la clase Base

En algunos casos, es posible que deseemos cambiar la forma en que un miembro de la clase base trabaja en la clase derivada.

Por ejemplo, la clase base ***Usuario*** incluye un método denominado ***CalcularID***.

```
public class Usuario
{
    public virtual string CalcularID()
    {
        return System.Guid.NewGuid().ToString();
    }
}
```

Debido a que ***CalcularID*** es un método virtual, podemos reemplazarlo en una clase derivada. Para reemplazar un método virtual en una clase derivada, creamos un método con la misma firma y le anteponemos la palabra clave ***override***.

El siguiente ejemplo, muestra como sobrescribir el método virtual en una clase derivada.

```
public class Administrador : Usuario
{
    public override string CalcularID()
    {
        return "A" + "-" + System.Guid.NewGuid().ToString();
    }
}
```

Podemos realizar el mismo proceso para reemplazar propiedades, indexadores y eventos. En cada caso, sólo se puede reemplazar un miembro de la clase base si el miembro está marcado como virtual en la clase base. No es posible reemplazar constructores.

También podemos reemplazar un miembro de la clase base mediante la palabra clave ***new***.

```
public class Administrador : Usuario
{
    public new string CalcularID()
    {
        return "A" + "-" + System.Guid.NewGuid().ToString();
    }
}
```

Cuando se utiliza la palabra clave ***override***, el método extiende el método de la clase base. Por el contrario, cuando se utiliza la palabra clave ***new***, el método oculta el método de la clase base. Esto causa sutiles pero importantes diferencias en la forma en que el compilador trata a la clase base y las clases derivadas.

Para obtener más información acerca de las diferencias del comportamiento entre ***override*** y ***new***, se recomienda visitar el siguiente enlace:

Knowing When to Use Override and New Keywords (C# Programming Guide)
https://msdn.microsoft.com/en-us/library/ms173153(v=vs.140).aspx

Sellando Miembros Reemplazados

Cuando en una clase se sobrescribe a un miembro de la clase base, podemos evitar que estos miembros remplazados sean a su vez remplazados en alguna clase que derive de la clase que ha remplazado los miembros. Para evitar que un miembro remplazado sea remplazado en clases derivadas, utilizamos la palabra clave **sealed**.

El siguiente ejemplo, muestra como sellar un miembro de clase que ha sido sobrescrito.

```
public class Administrador : Usuario
{
    sealed public override string CalcularID()
    {
        return "A" + "-" + System.Guid.NewGuid().ToString();
    }
}
```

Al sellar un miembro sobrescrito, forzamos a las clases que se derivan de esa clase a utilizar la nueva implementación en lugar de crear una nueva implementación. Esto puede ser útil cuando

necesitamos controlar el comportamiento de nuestras clases y asegurarnos de que las clases derivadas no intenten modificar la forma en que trabajan miembros específicos.

Sólo se puede aplicar el modificador ***sealed*** a un miembro si este es un miembro ***override***. Recordemos que los miembros están intrínsecamente sellados a menos que estén marcados como ***virtual***. En nuestro ejemplo, debido a que el método de la clase base está marcado como ***virtual***, los herederos son capaces de reemplazar el método a menos que lo sellemos en algún punto de la jerarquía de clases.

Invocando a los Constructores y Miembros de la Clase Base

En una clase derivada, podemos utilizar la palabra clave ***base*** para acceder a los métodos y constructores de la clase base. Esto es útil en los siguientes escenarios:

- Cuando sobrescribimos un método de la clase base, pero deseamos seguir ejecutando la funcionalidad del método de la clase base además de nuestra propia funcionalidad adicional.
- Cuando creamos un método y deseamos invocar a un método de la clase base como parte de la lógica del método.
- Cuando creamos un constructor y deseamos invocar a un constructor de la clase base como parte de la lógica de inicialización.
- Cuando queremos invocar a un método de la clase base desde un accesor de propiedad.

Invocando Constructores de la Clase Base desde una Clase Derivada

No es posible sobrescribir constructores en clases derivadas. Cuando creamos constructores en una clase derivada, los constructores invocarán automáticamente al constructor predeterminado de la clase base. Sin embargo, en algunas circunstancias, podríamos querer que los constructores invoquen a un constructor alterno de la clase base. En estos casos, podemos utilizar la palabra clave ***base*** en la declaración del constructor para especificar cual constructor de la clase base queremos invocar.

El siguiente ejemplo, muestra cómo invocar a constructores de la clase base.

```
public class Administrador : Usuario
{
    // Este constructor invoca implícitamente al constructor predeterminado
    // de la clase base. Esta declaración equivale implícitamente a:
    // public Administrador() : base()
    public Administrador()
    {
        // Implementación adicional
    }

    // Este constructor invoca explícitamente al constructor de la clase base.
    public Administrador(decimal sueldoDiario) :base(sueldoDiario)
    {
       // Implementación adicional
    }
    sealed public override string CalcularID()
    {
        return "A" + "-" + System.Guid.NewGuid().ToString();
    }
}
```

Como podemos ver, se debe invocar al constructor de la clase base desde la declaración del constructor. No se puede invocar al constructor de la clase base dentro del cuerpo del constructor. También es posible pasar argumentos por nombre al constructor de la clase base.

Invocando Métodos de la Clase Base desde una Clase Derivada

Es posible invocar métodos de la clase base desde el cuerpo de métodos o accesores de propiedad de la clase derivada. Para hacer eso, utilizamos la palabra clave ***base*** de la misma forma en que utilizamos una variable conteniendo la instancia de una clase.

El siguiente ejemplo, muestra cómo invocar métodos de la clase base.

```
public override string CalcularID()
{
    return "A" + "-" + base.CalcularID();

}
```

Es importante recordar que las reglas de herencia no aplican a miembros y clases estáticas. Por lo tanto, no podemos utilizar la palabra clave base dentro de un método estático.

Lección 2:
Extendiendo Clases del .NET Framework

El .NET Framework contiene miles de clases que proporcionan un amplio rango de funcionalidad. Es recomendable que, al crear nuestras propias clases, siempre que sea posible, estas sean creadas heredando de alguna clase del .NET Framework. Esto no solo reduce la cantidad de código que debemos escribir, también nos ayuda a asegurar que nuestras clases trabajen en una forma estandarizada.

El .NET Framework también nos permite crear ***métodos de extensión*** para agregar funcionalidad a tipos sellados del .NET Framework. Esto nos permite extender la funcionalidad de los tipos predefinidos, tales como la clase **String**, cuando la herencia no está permitida.

En esta lección, aprenderemos a extender los tipos del .NET Framework mediante el uso de herencia y métodos de extensión.

Objetivos de la lección

Al finalizar esta lección, los participantes contarán con las habilidades y conocimientos para:

- Crear clases que hereden de tipos del .NET Framework.
- Crear clases *Exception* personalizadas.
- Lanzar y atrapar excepciones personalizadas.
- Crear clases que hereden de tipos genéricos.
- Crear métodos de extensión para tipos del .NET Framework.

Heredando de Clases del .NET Framework

Existen más de 15,000 tipos públicos en el .NET Framework, todos ellos distribuidos en múltiples Assemblies y organizados de manera lógica en espacios de nombres. Aunque no todos estos tipos son clases extensibles, muchos de ellos lo son. Cuando deseamos desarrollar una clase, en la mayoría de los casos, existe una clase del .NET Framework que nos puede proporcionar una base para nuestro código.

Existen dos ventajas principales al crear una clase que herede de una clase del .NET Framework en lugar de crear la clase desde cero:

- **Reducir el tiempo de desarrollo**. Al heredar de una clase existente, se reduce la cantidad de lógica que tenemos que crear nosotros mismos.
- **Funcionalidad estandarizada**. De la misma forma que implementamos una interface, heredar de una clase base estándar significa que nuestra clase funcionará de una forma estandarizada. También podemos representar instancias de nuestra clase como instancias de la clase base, lo que hace que sea mucho más fácil para los desarrolladores utilizar nuestra clase junto a otros tipos que derivan de la misma clase base.

Las reglas de herencia también se aplican a las clases integradas del .NET Framework de la misma forma en que se aplican a las clases personalizadas:

- Podemos crear una clase que derive de una clase del .NET Framework, siempre y cuando la clase no esté sellada (***sealed***) o sea estática (***static***). Por ejemplo, en el caso de la clase ***System.IO.File***, esta es una clase estática por lo cual no podemos heredarla.
- Podemos reemplazar cualquier miembro de la clase base que esté marcado como ***virtual***.
- Si se hereda de una clase abstracta, debemos proporcionar implementaciones de todos los miembros abstractos.

Al crear una clase, debemos seleccionar una clase base que minimice la cantidad de codificación y personalización requerida. Si nos encontramos a nosotros mismos replicando la misma funcionalidad que se encuentra en las clases integradas del .NET Framework, probablemente deberíamos elegir una clase base más específica. Por otro lado, si encontramos que necesitamos sobrescribir muchos miembros, probablemente deberíamos elegir una clase base más general.

Por ejemplo, supongamos que deseamos crear una clase que almacene una lista lineal de valores. La clase debe permitir eliminar elementos duplicados de la lista. En lugar de crear una nueva clase ***List*** desde cero, podemos lograr esto, creando una clase que herede de la clase genérica ***List <T>*** y agregar simplemente un método para eliminar los elementos duplicados. Además, podemos tomar

ventaja del método ***Sort*** de la clase ***List<T>***. Si invocamos al método ***Sort***, los elementos duplicados aparecerán juntos en la colección, lo que puede hacer que sea más fácil de identificarlos y eliminarlos.

El siguiente ejemplo, muestra cómo extender la clase ***List<T>***.

```
class UniqueList<T> : List<T>
{
    public void RemoveDuplicates()
    {
        base.Sort();
        for (int i = this.Count - 1; i > 0; i--)
        {
            if (this[i].Equals(this[i - 1]))
            {
                this.RemoveAt(i);
            }

        }
    }
}
```

Al utilizar este enfoque, los consumidores de nuestra clase tendrán acceso a toda la funcionalidad proporcionada por la clase base ***List<T>***. También tendrán acceso al método adicional ***RemoveDuplicates*** que proporcionamos a nuestra clase derivada.

Creando Excepciones personalizadas

El .NET Framework contiene clases ***Exception*** integradas para representar las condiciones de error más comunes. Por ejemplo:

- Si invocamos un método con un valor de argumento nulo, y el método no puede manejar valores nulos en los argumentos, el método disparará una excepción de tipo ***ArgumentNullException***.
- Si intentamos dividir un valor numérico por cero, el motor de tiempo de ejecución lanzará una excepción de tipo ***DivideByZeroException***.
- Si intentamos recuperar un elemento indexado de una colección y el índice está fuera de los límites de la colección, el indexador lanzará una excepción de tipo ***IndexOutOfRangeException***.

La mayoría de las excepciones integradas del .NET Framework son definidas en el espacio de nombres **System**.

Siempre que sea posible, cuando necesitemos lanzar excepciones en nuestro código, debemos utilizar los tipos de excepción existentes en el .NET Framework. Sin embargo, puede haber circunstancias en las que queramos crear nuestros propios tipos de excepciones personalizadas.

¿Cuándo deberíamos crear un tipo de excepción personalizado?

Debemos considerar crear un tipo de excepción personalizada cuando:

- Los tipos de excepción existentes no representen adecuadamente la condición de error que estamos identificando.
- La excepción requiere una acción correctiva específica que difiere de cómo lo manejaríamos con los tipos de excepción integrados.

Recordemos que el propósito principal de los tipos ***Exception*** es permitir que nosotros manejemos condiciones de error específicas en formas específicas atrapando tipos de excepción específicos. El tipo ***Exception*** no está diseñado para comunicar los detalles precisos del problema. Todas las clases excepción incluyen una propiedad ***Message*** para este propósito. Por lo tanto, no debemos crear una clase de excepción personalizada sólo para comunicar la naturaleza de una condición de error. Debemos crear una clase excepción personalizada solo si necesitamos manejar esa condición de error de una manera distinta.

Creando tipos de excepción personalizados

Todas las clases ***Exception*** se derivan de la clase ***System.Exception***, debido a esto, un bloque ***Catch*** que atrapa excepciones del tipo ***System.Exception*** atrapará todas las excepciones y por lo tanto debemos atrapar primero excepciones más específicas. La clase ***System.Exception*** proporciona un conjunto de propiedades que podemos utilizar para proporcionar más detalle de la condición de error. Por ejemplo:

- La propiedad ***Message***, permite proporcionar más información acerca de lo sucedido mediante una cadena de texto.
- La propiedad ***InnerException***, permite identificar otra instancia ***Exception*** que causó la instancia actual.
- La propiedad ***Source***, permite especificar al elemento o la aplicación que provocó la condición de error.
- La propiedad ***Data***, permite proporcionar más información acerca de la condición de error a través de parejas llave-valor.

Cuando tengamos que crear un tipo de excepción personalizado, siempre que sea posible, debemos hacer uso de las propiedades existentes en lugar de crear nuestras propias propiedades alternas. A un alto nivel, el proceso de creación de una clase de excepción personalizada es la siguiente:

- Crear una clase que herede de la clase **System.Exception.**
  Una buena práctica recomendada al crear una excepción personalizada, es incluir la palabra ***Exception*** como sufijo del nombre de la clase. Anteriormente, era recomendado heredar de la clase ***ApplicationException*** para crear clases excepción personalizadas y, se destinaba a la clase ***SystemException***, para implementar excepciones exclusivamente por el CLR. Actualmente la recomendación es heredar de la clase ***System.Exception*** para crear clases ***Exception*** personalizadas.
- Mapear los constructores de la clase excepción personalizada con los constructores de la clase base.
- Agregar cualquier miembro que sea requerido

El siguiente ejemplo, muestra cómo crear una clase excepción personalizada.

```
class AmountNotAvailableException : Exception
{
    public AmountNotAvailableException()
    {
        // Este constructor invoca implícitamente al constructor de la clase base
    }

    public AmountNotAvailableException(string message) : base(message) { }
```

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

        public AmountNotAvailableException(SerializationInfo info,
            StreamingContext context) : base(info, context) { }

        public double Available { get; set; }

        public AmountNotAvailableException(string message, double available)
            : base(message)
        {
            this.Available = available;
        }
    }
```

Lanzando y capturando Excepciones personalizadas

Después de haber creado el tipo de excepción personalizado, podemos lanzar y atrapar excepciones personalizadas de la misma forma en que lanzamos y atrapamos cualquier otra excepción. Para lanzar una excepción personalizada, utilizamos la palabra clave ***throw*** y creamos una nueva instancia de la excepción personalizada.

El siguiente ejemplo, muestra la forma de lanzar una excepción personalizada.

```
double GetMoney(double amount)
{
    // implementar la lógica del método ....

    if (Available < amount)
    {
        // Lanzar la excepción personalizada
        throw new AmountNotAvailableException(
            "Fondos insuficientes", Available);
    }
    return Available;
}
```

Para atrapar la excepción, utilizamos un bloque ***try/catch***. Recordemos que siembre debemos intentar atrapar primero las excepciones más específicas y después las excepciones más generales, típicamente ***System.Exception*** al final.

El siguiente ejemplo, muestra cómo podemos atrapar una excepción personalizada.

```
void DoOperation()
{
    try
    {
        var monto = GetMoney(100);
    }
    // Atrapar la excepción personalizada.
    catch (AmountNotAvailableException ex)
    {
        Console.WriteLine(
            $"La cantidad máxima disponible es: {ex.Available}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error no esperado {ex.Message}");
    }
}
```

Heredando de tipos Genéricos

Al heredar de una clase genérica, debemos decidir cómo deseamos administrar los parámetros de tipo de la clase base. Podemos manejar los parámetros de tipo de dos formas:

- Dejar el parámetro de tipo de la clase base sin especificar.
- Especificar un argumento de tipo para la clase base.

Consideremos un ejemplo donde deseamos crear una clase de lista personalizada que herede de la clase ***List<T>***. Si dejamos el parámetro de tipo de la clase base sin especificar, debemos incluir el mismo parámetro de tipo en la declaración de la clase.

El siguiente ejemplo, muestra cómo heredar de un tipo base genérico sin especificar un argumento de tipo.

```
class ListaPersonalizada<T>:List<T>
{
}
```

En el ejemplo anterior, estamos heredando de una lista genérica y no estamos especificando el Tipo del parámetro de Tipo, esto es, el tipo con el que trabaje nuestra clase genérica puede ser cualquier tipo. Cuando se crea una instancia de la clase ***ListaPersonalizada*** y proporcionamos un argumento de tipo para T, el mismo argumento de tipo se aplica a la clase base. Por ejemplo, si la clase derivada recibe un parámetro ***int***, este tipo también es recibido por la clase Base.

Como alternativa, podemos especificar un argumento de tipo para el tipo base en la declaración de la clase. Cuando utilizamos este enfoque, todas las referencias al parámetro de tipo en el tipo base se reemplazan con el tipo que se ha especificado en la declaración de la clase.

El siguiente ejemplo, muestra cómo especificar un argumento de tipo para el tipo de la clase base.

```
class ListaPersonalizada : List<int>
{
}
```

En el ejemplo anterior, cuando se crean instancias de la clase ***ListaPersonalizada***, no es necesario especificar un parámetro para el tipo. Cualquier método o propiedad de la clase base que hace referencia al parámetro de tipo, está fuertemente tipado al tipo ***int***. Por ejemplo, el método ***List.Add*** sólo aceptará argumentos de tipo ***int***.

Si la clase base que se hereda contiene varios parámetros de tipo, podemos especificar argumentos de tipo para cualquiera de ellos. El punto importante a recordar es que debemos proporcionar un argumento de tipo o agregar parámetros de tipo a la declaración de la clase para cada parámetro de tipo en la clase base.

El siguiente ejemplo, muestra las diferentes formas en que podemos heredar de un tipo base con múltiples parámetros.

```
// Pasar todos los parámetros de tipo base en la clase derivada.
public class MiDiccionario<TKey, TValue> : Dictionary<TKey, TValue> { }

// Proporcionar un argumento para uno de los parámetros del tipo base y
// pasar el otro en la clase derivada.
public class MiDiccionario2<TValue> : Dictionary<int, TValue> { }

// Proporcionar argumentos para todos los parámetros del tipo base.
public class MiDiccionario3 : Dictionary<int, string> { }
```

Independientemente de cuantos parámetros de tipo incluye el tipo base, es posible agregar parámetros de tipo adicionales en la declaración de la clase derivada.

El siguiente ejemplo, muestra como agregar parámetros de tipo adicionales en la declaración de la clase derivada.

```
// Pasar el parámetro de tipo base en la clase derivada y
// agregar un parámetro de tipo adicional.
public class MiColeccion<T, U> : List<T> { }

// Proporcionar un argumento para el parámetro de tipo base
// y agregar un nuevo parámetro de tipo.
public class MiColeccion2<T> : List<int> { }

// Heredar de una clase no genérica pero agregar un parámetro de tipo.
public class MiColeccion3<T> : ArrayList { }
```

Creando Métodos de Extensión

En la mayoría de los casos, si deseamos extender la funcionalidad de una clase, utilizamos herencia para crear una clase derivada. Sin embargo, esto no siempre es posible. Muchos tipos integrados son sellados para evitar la herencia. Por ejemplo, no podemos crear una clase que extienda el tipo ***System.String*** para poder convertir un tipo ***string*** a ***int***.

Como una alternativa al uso de herencia para extender un tipo, podemos crear ***Métodos de Extensión***. Cuando creamos métodos de extensión, estamos creando métodos que podemos invocar sobre un tipo particular sin modificar realmente ese tipo particular. Un método de extensión es similar a un método estático. Para crear un método de extensión, debemos crear un método estático dentro de una clase estática. El primer parámetro del método especifica el tipo que queremos extender. Al preceder el parámetro con la palabra clave ***this***, indicamos al compilador que el método es un método de extensión de ese tipo.

El siguiente ejemplo, muestra cómo crear un método de extensión para el tipo ***System.String***.

```
namespace StringExtensionMethods
{
    public static class StringExtensions
    {
        public static int? ToInt(this string valor)
        {
            int Entero;
            int? Resultado = null;
            if (int.TryParse(valor, out Entero))
            {
                Resultado = Entero;
            }
            return Resultado;
        }
    }
}
```

Para utilizar un método de extensión, debemos importar explícitamente el espacio de nombres que contiene el método de extensión mediante el uso de una directiva ***using***.

```
using StringExtensionMethods;
```

Después de esto, podemos invocar el método de extensión como si fuese un método de instancia en el tipo que extiende.

```
string Numero = "123";
int? i = Numero.ToInt();
Console.WriteLine(i);
```

Introducción a C#

Apéndice A.
Nuevas características en C# 6

Introducción

C#, pronunciado en inglés como “C sharp”, es un lenguaje de programación diseñado para construir una gran variedad de aplicaciones que se ejecutan en el .NET Framework. C# es simple, poderoso, de tipos seguros y orientado a objetos. Las muchas mejoras que ha tenido C# permiten un rápido desarrollo de aplicaciones al mismo tiempo que mantiene la elegancia de los lenguajes de estilo C.

Visual C# es la implementación de Microsoft del lenguaje C#. La siguiente tabla resume las distintas características que han sido incorporadas a Visual C# así como las versiones de Visual Studio en las que se hicieron disponibles.

Versión	Versión de Visual Studio	Principales características
C# 1	Visual Studio .NET 2002	Primera versión.
C# 1.1	Visual Studio .NET 2003	Directiva #line y Comentarios de documentación XML.
C# 2	Visual Studio .NET 2005	Métodos anónimos, Tipos Genéricos, Tipos Anulables, Iteradores/yield, Clases estáticas, Covarianza y Contravarianza para delegados.
C# 3	Visual Studio .NET 2008	Inicializadores de Objetos y Colecciones, Expresiones Lambda, Métodos de Extensión, Tipos Anónimos, Propiedades Auto-implementadas, LINQ, inferencia de tipos (var).
C# 4	Visual Studio .NET 2010	Dynamic, Argumentos nombrados, Parámetros opcionales, covarianza y contravarianza para tipos Genéricos.
C# 5	Visual Studio .NET 2012	Async, Await, Atributos para obtener información de Invocadores: CallerMemberName, CallerFilePath, CallerLineNumber.
	Visual Studio .NET 2013	Corrección de algunos errores, mejoras en el rendimiento, versión preliminar de la plataforma .NET Compiler (Roslyn).
C# 6	Visual Studio .NET 2015	Métodos de extensión Add en Inicializadores de Colecciones, Inicializadores de Índice (Index initializers), Expresiones Nameof, Operador Null-Conditional, Interpolación de Cadenas (String interpolation), using static, await en bloques catch y finally, Exception filters, Inicializadores para propiedades Auto-Implementadas, Propiedades Auto-implementadas de sólo lectura, Miembros con una Expresión como cuerpo.

Métodos de extensión Add en Inicializadores de Colecciones

Cuando se implementaron por primera vez los inicializadores de colecciones en C#, los métodos **Add** que se tenían que invocar, no podían ser métodos de extensión, tenían que ser métodos de instancia. En Visual C# 6, los métodos **Add** de extensión ya son vistos por los inicializadores de objetos.

Por ejemplo, el siguiente código en C# 5 no compila mientras que en C# 6 si compila.

```
class Program
{
    static void Main()
    {
        Queue Q = new Queue { 5, 10, 15 };
    }
}

static class ExtensionMethods
{
    public static void Add(this Queue collection, object item)
    {
        collection.Enqueue(item);
    }
}
```

Inicializadores de Índice (Index initializers)

En C# 6 se ha agregado una sintaxis nueva para para inicializar objetos permitiendo establecer valores a llaves a través de cualquier Indizador (Indexer) que el objeto implemente. Con esta nueva característica, ahora podemos inicializar elementos específicos de una colección que implemente Indizadores, tal como un objeto **Dictionary**.

El siguiente ejemplo muestra el uso de esta nueva característica a través de un objeto **Dictionary**.

En C# 5.

```
// Insertamos elementos llave-valor
var SpanishDays = new Dictionary<byte, string>
{
    {2, "Lunes"},
    {3, "Martes"},
    {4, "Miércoles"},
    {5, "Jueves"},
    {6, "Viernes"}
};
```

En C# 6.

```
// Insertamos los valores de las llaves 2, 3, 4, 5, 6
var SpanishDays = new Dictionary<byte, string>
{
    [2] = "Lunes",
    [3] = "Martes",
    [4] = "Miércoles",
    [5] = "Jueves",
    [6] = "Viernes"
};
```

El siguiente ejemplo muestra una clase **Inventory** que implementa un Indizador.

```
class Inventory
{
    Dictionary<string, int> Products =
        new Dictionary<string, int>();
    public int this[string index]
    {
        get { return Products[index]; }
        set { Products[index] = value; }
    }
}
```

En C# 5 podemos inicializar y agregar elementos a una instancia de **Inventory** de la siguiente manera. Nótese que no es posible Inicializar y agregar elementos en la misma sentencia.

```
// Creamos la Instancia
var UnitsInStock = new Inventory();
// Insertamos elementos
UnitsInStock["Azucar"] = 5;
UnitsInStock["Leche"] = 10;
UnitsInStock["Frijol"] = 100;
```

En C# 6 podemos inicializar y agregar elementos a una instancia de **Inventory** de la siguiente manera.

```
// Creamos e insertamos en la misma sentencia
var UnitsInStock = new Inventory
{
    ["Azucar"] = 5,
    ["Leche"] = 10,
    ["Frijol"] = 100
};
```

Expresiones Nameof

A través del operador **nameof**, C# 6 nos permite obtener una cadena con el nombre de una Variable, Clase, Método, Parámetro, Propiedad o incluso el nombre de un atributo. La cadena con el nombre obtenido no contiene el espacio de nombres correspondiente.

El siguiente ejemplo nos permite obtener el nombre de un parámetro como cadena.

```
static void GetParameterName(int productID)
{
    string Name = nameof(productID);
    Console.WriteLine(Name);
}
```

En este ejemplo, la Consola mostraría:

productID

Un ejemplo práctico de la utilidad de esta característica podemos verlo cuando desarrollamos utilizando el patrón MVVM y necesitamos notificar que el valor de una propiedad ha cambiado. Para realizar esta notificación es necesario proporcionar el nombre de la propiedad como cadena. Podemos especificar el nombre de la cadena con código duro pero eso nos traería resultados inesperados si se nos ocurre modificar el nombre de la propiedad y no modificamos la cadena en código duro.

Con el operador **nameof**, el código podría quedar de la siguiente manera.

```
public decimal UnitPrice
{
    set
    {
        // ...
        RaisePropertyChanged(nameof(UnitPrice));
    }
}

public void RaisePropertyChanged(string propertyname)
{
    Console.WriteLine(propertyname);
}
```

En este ejemplo, la Consola mostraría:

UnitPrice

Es importante mencionar que el operador **nameof** no obtiene el espacio de nombres, únicamente el identificador. Por lo tanto, las siguientes sentencias mostrarían en la Consola el mismo resultado:

Main

```
Console.WriteLine(nameof(_03.Program.Main));
Console.WriteLine(nameof(Main));
```

Proporcionar al operador **nameof** el espacio de nombres junto con el identificador, sólo le indica a **nameof** el lugar donde se encuentra el identificador. Esto es bastante útil cuando tenemos el mismo identificador en distintos espacios de nombres.

Otro ejemplo útil del operador **nameof** es en aplicaciones ASP.NET MVC donde tenemos que especificar los nombres como cadena de los Controladores y métodos de acción como en el siguiente ejemplo.

En C# 5 utilizamos cadenas de texto para especificar nombres de Controladores y Acciones.

```
public ActionResult Index()
{
    return RedirectToAction("Index", "Login");
}
```

En C# 6 podemos utilizar el operador **nameof** evitando resultados inesperados al cambiar los nombres de los Controladores o Acciones ya que el compilador nos avisaría al compilar la aplicación.

```
public ActionResult Index()
{
    return RedirectToAction(
        nameof(Controllers.AccountController.Login),
        nameof(Controllers.AccountController).Replace("Controller",""));
}
```

En aplicaciones ASP.NET, para que el operador **nameof** sea reconocido en las Vistas durante la parte de compilación en tiempo de ejecución, es necesario habilitar la compilación **Roslyn** en ASP.NET instalando el siguiente paquete NuGet en la aplicación Web ASP.NET:

CodeDOM Providers for .NET Compiler Platform (“Roslyn”) NuGet package

Para instalar el paquete en el proyecto ASP.NET, podemos escribir lo siguiente desde **Package**

Manager Console en Visual Studio.

install-package Microsoft.CodeDom.Providers.DotNetCompilerPlatform

Para mayor información sobre la forma de habilitar la compilación Roslyn en aplicaciones ASP.NET, pueden consultar el siguiente enlace:

http://blogs.msdn.com/b/webdev/archive/2014/05/12/enabling-the-net-compiler-platform-roslyn-in-asp-net-applications.aspx

Operador Null-Conditional

El operador **Null-Conditional** puede ser representado de dos formas:

- Mediante un símbolo de interrogación con un punto para evaluar si un objeto es null antes de acceder a sus miembros: **?.**
- Mediante un símbolo de interrogación y un corchete cuadrado de apertura para evaluar si un Tipo indexado es **null** antes de acceder a uno de sus elementos utilizando un índice: **?[**

Con el operador **Null-Conditional** podemos verificar si una expresión es **null** antes de intentar acceder a alguno de sus miembros (**?.**) o antes de acceder a uno de sus elementos indexados (**?[**). Este operador nos ayuda a escribir menos código para manejar la verificación de **null**, especialmente al acceder a los miembros de las estructuras jerárquicas de datos.

Si tuviéramos por ejemplo el siguiente código:

```
string Controller = null;
//...
//... Asignar un valor a Controller
//...
string ControllerName =
    Controller.Replace("Controller", "");
```

Al ejecutar la última instrucción, se dispararía la excepción **NullReferenceException** si **Controller** sigue siendo **null**.

En C# 5, para evitar la excepción podríamos escribir el siguiente código:

```
string ControllerName;
if(Controller==null)
{
    ControllerName = null;
}
else
{
    ControllerName =
        Controller.Replace("Controller", "");
}
```

En C# 6, podemos escribir el siguiente código que es el equivalente al código anterior:

```
string ControllerName =
    Controller?.Replace("Controller", ""); ;
```

El operador **Null-Conditional** también nos permite manejar valores **null** cuando accedemos a elementos indexados.

Por ejemplo, si tenemos la siguiente declaración:

```
byte[] Numbers = null;
//...
// Instanciar el arreglo
//...
```

El siguiente código nos permite almacenar en una variable el valor del primer elemento de un arreglo de bytes cuando el arreglo no es **null**. Si el arreglo es **null**, la variable que almacena el primer elemento será **null** igualmente.

En C# 5:

```
byte? FirstElement;
if(Numbers==null)
{
    FirstElement = null;
}
else
{
    FirstElement = Numbers[0];
}
```

En C# 6 utilizando el operador **Null-Conditional:**

```
byte? FirstElement = Numbers?[0];
```

El operador **Null-Conditional** tiene un comportamiento de Cortocircuito (short-circuiting), lo que significa que si una operación en la secuencia de acceso a miembros o a operaciones con índices devuelve **null**, entonces el resto de la cadena de ejecución se detiene.

Supongamos ahora que tenemos la siguiente declaración:

```
Customer[] Customers = null;
//...
//... Agregar Customers
//...
int? Count;
```

Queremos que si el arreglo **Customers** es **null** o el elemento **Customer[0]** es **null** o si **Customer[0].Orders** es **null**, entonces la variable **Count** reciba **null**, de lo contrario, la variable **Count** debe recibir el número de elementos de la colección **Customers[0].Orders**.

El código en C# 5 quedaría de la siguiente forma:

```
if(Customers==null ||
    Customers[0]==null ||
    Customers[0].Orders==null)
{
    Count = null;
}
else
{
    Count = Customers[0].Orders.Count();
}
```

El siguiente código C# 6 es equivalente al código anterior. Podemos notar la diferencia en el número de líneas de código necesarias para implementar la solución:

```
int? Count = Customers?[0]?.Orders?.Count();
```

En el ejemplo anterior, si tuviéramos en la expresión otras operaciones de menor precedencia, estas continuarían su ejecución como en el siguiente caso:

```
int Count2 = Customers?[0]?.Orders?.Count() ?? 0;
```

En este caso estamos utilizando el operador **null-coalescing** (**??**) dando como resultado que si la expresión **Customers?[0]?.Orders?.Count()** devuelve **null**, entonces la variable **Count2** recibe **0**, de lo contrario, la variable **Count2** recibe el valor **Customers[0].Orders.Count()**.

Otro uso bastante útil del operador **Null-Conditional** es la invocación de delegados en un hilo seguro. El siguiente ejemplo muestra el código C# 5 común utilizado para disparar el evento **PropertyChanged** en una clase ViewModel que implementa la Interface **INotifyPropertyChanged**.

```
void RaisePropertyChanged(string propertyName)
{
    var Subscribers = PropertyChanged;
    if (Subscribers != null)
    {
        Subscribers(this,
        new PropertyChangedEventArgs(propertyName));
    }
}
```

El código equivalente en C# 6 quedaría de la siguiente forma.

```
void RaisePropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}
```

Esta nueva forma es de hilo seguro (Thread-safe) ya que el compilador genera el código para evaluar **PropertyChanged** una sola vez almacenando el resultado en una variable temporal.

Es necesario invocar explícitamente el método **Invoke** debido a que no existe una sintaxis **null-conditional** para invocar delegados.

Interpolación de Cadenas (String interpolation)

El concepto de Interpolar se define como la acción de poner algo entre otras cosas. También se define como la acción de intercalar palabras o frases dentro de un texto.

En C# 6 podemos utilizar expresiones de interpolación de cadenas para construir nuevas cadenas. Una expresión de cadena interpolada es similar a una plantilla de cadena que contiene expresiones.

Supongamos que tenemos el siguiente código:

```
string ProductName = "Azucar";
double UnitPrice = 12.45;
```

Queremos enviar a la Consola el siguiente texto:

El producto Azucar tiene un precio de 12.45

En C# 5 podemos formatear la salida con la técnica de argumentos de la siguiente manera:

```
Console.WriteLine(
    "El producto {0} tiene un precio de {1}",
    ProductName, UnitPrice);
```

En C# 6 podemos formatear la salida utilizando una cadena interpolada de la siguiente manera:

```
Console.WriteLine(
    $"El producto {ProductName} tiene un precio de {UnitPrice}");
```

Podemos ver que estamos incrustando las expresiones **ProductName** y **UnitPrice** dentro de la cadena**.**

Cada vez que se ejecuta el código con la cadena interpolada, C# crea una nueva cadena a partir de la cadena interpolada remplazando las expresiones por su representación ToString. Una cadena interpolada es muy fácil de entender en comparación con el formateo de cadenas que utilizan argumentos tales como String.Format.

Para incluir el carácter “{“ o “}” en una cadena interpolada, utilizamos llaves dobles como en el siguiente ejemplo.

```
Console.WriteLine(
    $"Nombre del producto: {{{ProductName}}}");
```

El resultado es el siguiente:

Nombre del producto: {Azucar}

Al igual que con **String.Format**, podemos especificar alineación y formatos como en el siguiente ejemplo.

```
Console.WriteLine(
    $"{ProductName,20} tiene un precio de {UnitPrice:C2} pesos");
Console.WriteLine(
    $"20 kilos equivalen a {UnitPrice*20:C0} pesos");
```

El código anterior produce la siguiente salida.

Azucar tiene un precio de $12.45 pesos

20 kilos equivalen a $249 pesos

Nótese que en la primera instrucción, el nombre del producto ocupa 20 posiciones.

Una cadena interpolada puede incluso incluir expresiones cadena como en el siguiente ejemplo.

```
int n = 40;
var s =
    $"El número {n} es {(n%2==0 ? "Par": "Impar")}";
```

Debemos notar que la expresión condicional está delimitada por paréntesis.

En este ejemplo, el valor final de la variable **s** sería:

El número 40 es Par

using static

La característica **using static** permite que todos los miembros estáticos accesibles de una clase, puedan ser importados para poder acceder a ellos sin la necesidad de especificar la clase en la que se encuentran implementados.

El siguiente ejemplo muestra el uso de la característica **using static** para acceder a los miembros estáticos de la clase **Console** sin necesidad de especificar el nombre de la clase **Console**.

```
using static System.Console;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Write("¿Cuál es tu nombre?");
            var Name = ReadLine();
            WriteLine($"Hola {Name}!");
        }
    }
}
```

Esta característica es muy útil cuando tenemos un conjunto de funciones dentro de una clase que utilizamos frecuentemente.

La característica **using static**, permite también especificar directamente los valores nombrados de una enumeración como en el siguiente ejemplo.

```
using static Status;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Status s = On;
            WriteLine($"Current status {s}");
        }
    }

}

enum Status
{
    On,
    Off
```

```
}
```

Métodos de Extensión

Los métodos de extensión son métodos estáticos diseñados para ser utilizados como métodos de instancia. El siguiente código muestra una clase con dos métodos estáticos, uno de ellos es un método de extensión de un tipo **string**.

```
public static class Helper
{
    // Método estático
    public static int GetRamdom()
    {
        return new System.Random().Next(1, 1000);
    }

    // Método de extensión de un tipo string
    public static int? ToInt(this string source)
    {
        int? Result = null;
        int ToConvert;
        if(int.TryParse(source, out ToConvert))
        {
            Result = ToConvert;
        }
        return Result;
    }
}
```

La característica **using static**, no importa a los métodos de extensión de la misma forma en que lo hace con los métodos estáticos que no son de extensión.

Tomando como base el código anterior de la clase **Helper**, la siguiente instrucción generaría el error de compilación "**The name 'ToInt' does not exist in the current context**".

```
using static Helper;

namespace _06
{
    class Program
    {
        static void Main()
        {
            var n2 = ToInt(n);
        }
    }
}
```

Sin embargo, las siguientes instrucciones son completamente válidas.

```
using static System.Console;
using static Helper;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Write("Proporciona un número entero: ");
            string n = ReadLine();

            // Se hace disponible el método de extensión
            int? Number = n.ToInt();

            // El método de extensión puede utilizarse
            // especificando la Clase Helper
            int? Number3 = Helper.ToInt(n);

            // Los métodos estáticos se hacen disponibles
            // sin especificar la clase Helper
            var r = GetRamdom();
        }
    }
}
```

Uso del operador await en bloques catch y finally

En C# 5 y versiones anteriores no era posible realizar llamadas a métodos asíncronos dentro de los bloques **catch** y **finally**. Por ejemplo, el siguiente código generaba los errores de compilación "**Cannot await in the body of a catch clause**" y "**Cannot await in the body of a finally clause**".

```
static async Task<bool> DoSomething()
{
    var L = new Log();
    bool Result=true;
    try
    {
        await L.OpenAsync();
    }
    catch (Exception ex)
    {
        var Status =
            await L.WriteAsync(
            "Error encontrado: " + ex.Message);
        Result = false;
    }
    finally
    {
        var Status =
            await L.CloseAsync();
    }
    return Result;
}
```

Este mismo código ahora ya compila y se ejecuta sin problemas en C# 6.

Filtros de Excepción (Exception filters)

Los Filtros de Excepción son una capacidad del CLR disponible en Visual Basic y F# pero que no se encontraba disponible en C# hasta ahora con C# 6.

Supongamos que tenemos la necesidad de invocar dos métodos que podrían generar una excepción del tipo **HttpException.** En caso de que el código http de error sea 400 o 401 queremos registrar las excepciones a la bitácora de error de la aplicación e ignorar la excepción. En caso de que el código de error http sea distinto a 400 y 401, debemos dejar pasar la excepción al método que haya invocado al código que queremos crear.

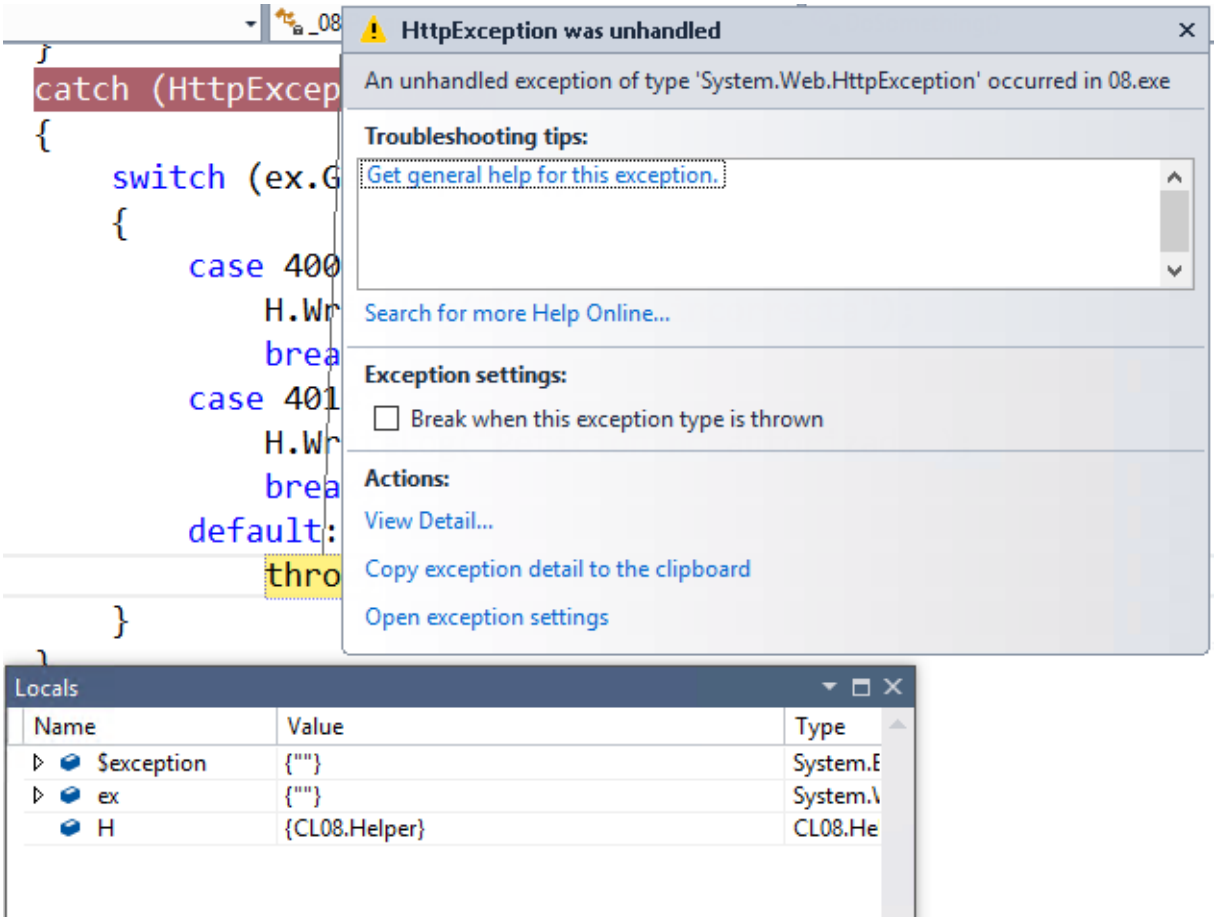
El código propuesto es el siguiente.

```
static void DoSomething()
{
    var H = new Helper();
    try
    {
        string Message = "Hello, World";
        H.SendHttpMessage(Message, true);
        string Html = H.GetHttpResource("/Products", true);
    }
    catch (HttpException ex)
    {
        switch (ex.GetHttpCode())
        {
            case 400:
                H.WriteLog("Petición incorrecta");
                break;
            case 401:
                H.WriteLog("Petición no autorizada");
                break;
            default:
                throw;
        }
    }
}
```

En el código anterior, podemos notar que estamos invocando a los métodos **SendHttpMessage** y **GetHttpResource** de un objeto **Helper**. Al invocar a estos métodos podemos recibir una excepción del tipo **HttpException** que estamos atrapando. Si los códigos de error son el 400 o 401 escribimos una entrada en el log de la aplicación, en caso contrario volvemos a lanzar la excepción.

Cuando se genere una excepción con código diferente a 400 o 401, la ejecución se detendrá en la sentencia **throw** y si observamos la ventana **Locals**, veremos que las variables **Message** y **Html** se

encuentran fuera de alcance por lo que no aparecerán. No veremos a simple vista cuál fue la línea que originó la excepción.



Con C# 6, podemos reescribir el código anterior utilizando un Filtro de Excepción de la siguiente manera:

```
static void DoSomething()
{
    var H = new Helper();
    try
    {
        string Message = "Hello, World";
        H.SendHttpMessage(Message, true);
        string Html = H.GetHttpResource("/Products", true);
    }
    catch (HttpException ex) when (WriteLog(ex.GetHttpCode(), H))
    {
    }
}
```

Cuando se genere una excepción del tipo **HttpException**, se evaluará la expresión **when()**. Si el resultado de la evaluación es **true** entonces se ejecutará el código del bloque **catch**, lo que significa que la excepción será atrapada. Si el resultado es **false**, el código del bloque catch **NO** será ejecutado, lo que significa que la excepción no será atrapada y por consiguiente será pasada al código que haya invocado al método **DoSomething**.

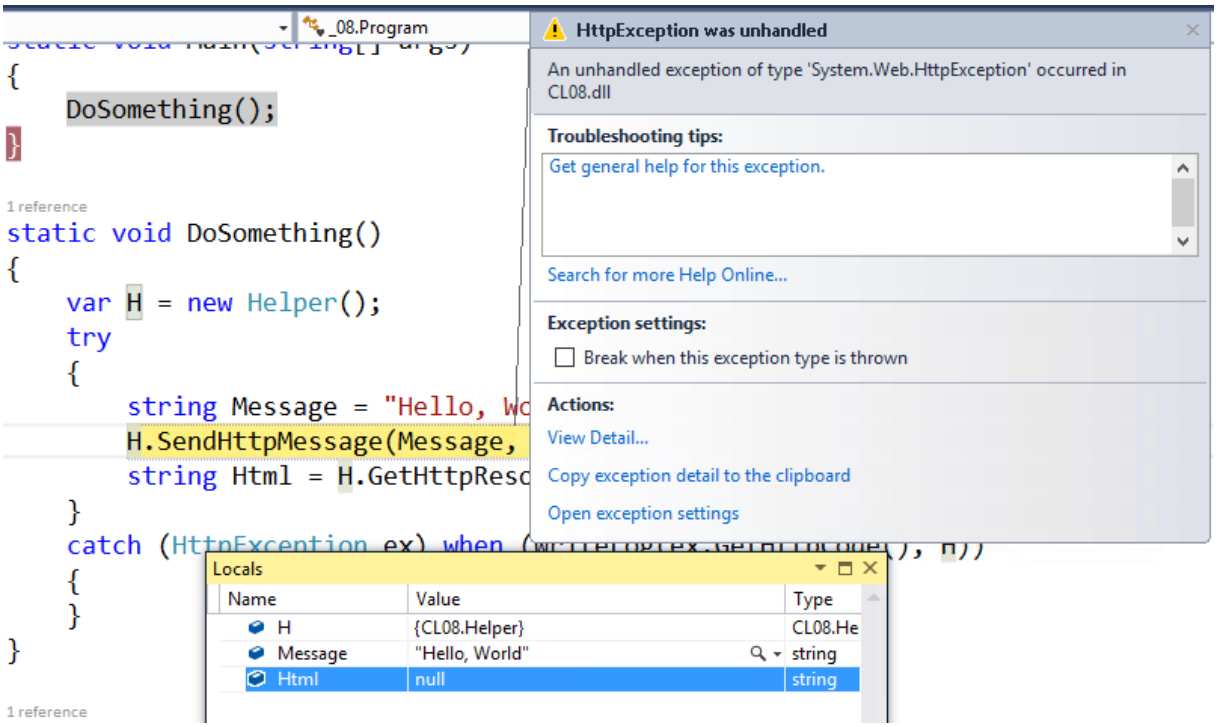
En caso de que la excepción no haya sido atrapada, se seguirán evaluando otros bloques **catch** en caso de que existan.

El código del método **WriteLog** quedaría de la siguiente manera.

```
static bool WriteLog(int httpCode, Helper H)
{
    bool ExecuteCatch = true;
    switch (httpCode)
    {
        case 400:
            H.WriteLog("Petición incorrecta");
            break;
        case 401:
            H.WriteLog("Petición no autorizada");
            break;
        default:
            ExecuteCatch = false;
            break;
    }
    return ExecuteCatch;
}
```

Podemos notar que si el código http no es 400 o 401, la variable **ExecuteCatch** obtiene el valor **false** lo que significa que no se ejecutará el bloque **catch**.

Cuando se genere una excepción con código diferente a 400 o 401, la ejecución se detendrá en la línea que causó la excepción y si observamos la ventana **Locals**, veremos que las variables **Message** y **Html** se encuentran ahí con sus valores actuales.



Los filtros de excepción son preferibles en lugar de atrapar y volver a lanzar la excepción ya que dejan el **Stack** intacto. Si posteriormente se dispara la excepción, podemos ver el lugar donde se originó la excepción en lugar de ver únicamente el lugar donde se volvió a disparar dicha excepción.

Es algo común y aceptado “abusar” del uso de filtros de excepción para invocar métodos que realicen tareas tales como guardar mensajes en el log de la aplicación sin manejar la excepción. En esos casos, el filtro de excepción (el método invocado) devolverá siempre **false**.

Inicializadores para propiedades Auto-Implementadas

C# 6 viene con una nueva característica que nos permite establecer el valor de una propiedad Auto-Implementada durante su declaración de la misma forma en que lo hacemos con los campos o variables.

Los siguientes son ejemplos de esta característica:

```
public int ProductID { get; set; } = 1;
public string ProductName { get; set; } = "Azucar";
public Double UnitPrice { get; set; } = 12;
```

Es posible también utilizar expresiones que hagan referencia a Constantes o miembros estáticos como en el siguiente ejemplo:

```
const int TAX = 16;

public Double UnitPrice2 { get; set; } = 12 * TAX;
public Double UnitPrice3 { get; set; } = 12 * GetTAX();
public string Location { get; set; } =
    ConfigurationManager.AppSettings["Location"];

static int GetTAX()
{
    // Algunos calculos
    return
    int.Parse(ConfigurationManager.AppSettings["TAX"]??"10");

}
```

El inicializador inicializa directamente el campo asociado a la propiedad (backing field). No lo hace a través del **accesor set** de la propiedad.

Los inicializadores son ejecutados en el orden en el que son escritos.

Los inicializadores de propiedades auto-implementadas no pueden referenciar a **this** ni a otros miembros de instancia ya que estos son ejecutados antes de que el objeto sea inicializado apropiadamente.

Propiedades Auto-implementadas de sólo lectura

Las propiedades Auto-implementadas ahora pueden ser declaradas sin un descriptor de acceso **set** como se muestra en el siguiente ejemplo:

```
public int ProductID { get; }
```

La variable (backing field) asociada a la propiedad es implícitamente declarada como **readonly**.

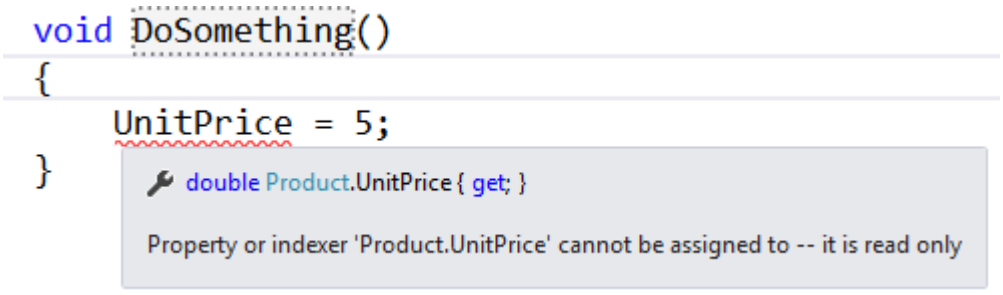
Podemos inicializar la propiedad directamente al declararla o bien en el constructor como se ve en el siguiente ejemplo:

```
public double UnitPrice { get; } = 10;
public double UnitPrice2 { get; }
public Product(int tax)
{
    UnitPrice2 = UnitPrice * tax;
}
```

Cualquiera que sea la forma en que inicialicemos la propiedad, el valor es asignado directamente a la variable (backing field) asociada a la propiedad.

Esta nueva característica hace que las propiedades auto-implementadas ahora puedan ser inmutables ya que al no tener un descriptor de acceso **set** y una variable backing field conocida, sólo podamos asignarle un valor durante su declaración o en el constructor.

El siguiente código mostraría un error al compilar.



Miembros con una Expresión como cuerpo (Expression-bodied function members)

Con C# 6 podemos implementar el cuerpo de Métodos, Propiedades, Indizadores o Sobrecarga de Operadores con la misma sintaxis que utilizamos al trabajar con expresiones lambda tal como se muestra en el siguiente ejemplo:

```
static void WriteMessage(string message) => Console.WriteLine(message);
```

El efecto es exactamente el mismo que tiene un método cuyo cuerpo es un bloque de código con una simple instrucción.

```
static void WriteMessage(string message)
{
    Console.WriteLine(message);
}
```

El siguiente ejemplo muestra un Método con una expresión como cuerpo:

```
public Complex Add(Complex b) =>
    new Complex(this.Real + b.Real, this.Imaginary + b.Imaginary);
```

El ejemplo anterior representa a un Método con sólo una instrucción **return** como se muestra a continuación:

```
public Complex Add(Complex b)
{
    return
        new Complex(this.Real + b.Real, this.Imaginary + b.Imaginary);
}
```

Propiedades e Indizadores de sólo lectura también pueden tener una expresión como cuerpo de su descriptor de acceso **get** tal y como se muestra en el siguiente ejemplo:

```
public string AsString => $"{Real} + {Imaginary}i";

public double this[byte index] =>
    index==0 ? Real:Imaginary;
```

Podemos notar que no especificamos la palabra clave **get**, esta es inferida por el uso de la sintaxis de la expresión como cuerpo.

En sobrecarga de operadores o sobre-escritura de miembros también podemos utilizar esta sintaxis tal y como se muestra en el siguiente ejemplo:

```
public static Complex operator +(Complex A, Complex B)=>A.Add(B);
public static implicit operator string(Complex A)=>A.AsString;
public override string ToString() => AsString;
```