Programación II

Table of contents:

- Características de C#
- Gramática de C#
  - Sintaxis de C#
  - Vocabulario de C#
- Trabajando con variables
  - Operadores de asignación
  - Constantes
  - Nombres de variables y atributos
- Common Type System
- Tipos de valor y tipos de referencia
  - Categorías de tipos
- Aliases
- Literales
- Caracteres
- Strings
- Tipos numéricos
  - Operadores aritméticos
  - Operadores de asignación
  - Enteros
  - Punto flotante
  - Notación binaria y hexadecimal
- Tipos booleanos
  - Operadores de igualdad
  - Operadores de comparación
- Tipo object
- Tipo dynamic
- Inferencia de tipos
- Valores por defecto
- Tamaño de tipos
- Conversiones de tipos de datos
  - Implicitas
  - Explicitas
- Operadores lógicos
  - Negación
  - Operador AND - Conjunción lógica
  - Operador OR - Disyunción lógica
  - Operador XOR - Disyunción exclusiva
- Operadores lógicos condicionales
- Sentencias de selección
  - Sentencia if
  - Sentencia switch

Programa

⚠ **SITIO EN CONSTRUCCIÓN**

Estamos trabajando en el desarrollo de las secciones, por lo que **no todo el contenido se encontrará disponible desde el inicio del cuatrimestre**.

Les pedimos paciencia y que nos ayuden a encontrar posibles errores.

Primer parcial

Clase 01 - Introducción a .NET y C#

Segundo parcial

*Last updated on **8/23/2021** by **mauriciocerizza***

On this page

Antes de empezar

Índice

Todas las secciones arrancan con un índice con enlaces a las distintas páginas de apuntes y ejercicios del tema.

En el índice también se encontrarán referencias a la bibliografía y material de consulta adicional.

Apuntes teóricos

Cada sección cuenta con su conjunto de apuntes teóricos que son el material de apoyo oficial de la materia.

Estos se complementan con los dictados de clases. En los parciales se podrán evaluar tanto el contenido de los apuntes como el dado en clase.

Convenciones

- Se utilizará **negrita** para remarcar conceptos clave o nueva terminología técnica.
- Se utilizará `énfasis` para hacer referencia a palabras del código, extensiones, nombres de carpetas, etc.
- Se utilizará *itálica* para marcar palabras a las que se les quiera dar realce.

Descarga en PDF

Si lo necesitaran, pueden descargar los apuntes y ejercicios en formato PDF.

Descargar apuntes y ejercicios en PDF

Ejercicios prácticos

Todas las secciones cuentan con ejercicios de resolución práctica como ejercitación para el parcial de Laboratorio de Computación II y los trabajos prácticos.

Al final de cada ejercicio puede llegar a estar habilitado el enlace a su resolución en código y/o video.

Categorías de ejercicios prácticos

Los mismos se encuentran categorizados según la siguiente tabla:

Categorías	Prefijo	Descripción
Introductorio	I##	Se enfocan en el contenido de la sección, no dependen de ejercicios previos y vienen acompañados de un diagrama de clase o descripción paso a paso de lo que hay que hacer.
Análisis funcional	A##	Plantean un requerimiento funcional y no especifican en profundidad cómo resolverlo. Sirven para ejercitar la creatividad junto con las capacidades de análisis y diseño.
Corrección de errores	E##	Son ejercicios de detección y corrección de errores, una tarea habitual en el ambiente profesional.
Consolidación	C##	Sirven para ejercitar el tema de la sección en conjunto con los aprendidos en clases anteriores. Pueden tener dependencias con ejercicios previos.
Investigación	R##	Para resolverlos se requerirá investigar contenido no abarcado por el temario. Son desafiantes y refuerzan las habilidades de búsqueda de soluciones en internet, aprendizaje auto-didacta e interpretación de documentación.

Recomendaciones sobre el uso de los ejercicios prácticos

- Consulte a su docente cuáles ejercicios le recomienda realizar y en qué orden.
- Cuantos más ejercicios haga, más aprenderá. Lleve la guía al día.
- Ante una duda o problema primero investigue en internet y trate de resolverlo por su cuenta. Como segunda instancia puede preguntar a un docente o en el canal de Slack, detallando cuál es la situación y qué intentó hacer previamente.

Cuestionarios

Todas las secciones finalizan con un cuestionario para reforzar lo aprendido y como ejercitación para el parcial teórico. Recomendamos ir realizando los cuestionarios semana a semana.

Recomendaciones sobre el uso de los cuestionarios

1. Repase la clase en cuestión.
2. Realice la ejercitación práctica.
3. Estudie como si fuera a dar un parcial sobre ese tema específico.
4. Responda el cuestionario tratando de no ayudarse con ninguna otra fuente que sus propios conocimientos.
5. Si lo intentó y aun así no puede responder la pregunta, puede ayudarse del material teórico o investigar.
6. Verifique sus respuestas contrastando con los apuntes y el material teórico.
7. Refuerce sus conocimientos y respuestas en clase. De ser necesario, consulte sus dudas con el profesor.
8. Repase lo aprendido antes del parcial.

*Last updated on **8/21/2021** by **mauriciocerizza***

On this page

Índice - Introducción a .NET y C#

Resumen

En esta asignatura haremos uso de la plataforma de desarrollo .NET y del lenguaje de programación C# para entender conceptos y prácticas que son comunes a muchas otras herramientas que se utilizan al desarrollar software.

Se apuntará a introducir las principales características de la plataforma y del lenguaje para que puedan alcanzar un entendimiento general sobre las herramientas que utilizaremos durante la cursada.

Apuntes

Ejercicios

Introducción

I03 - Los primos

I04 - Un número perfecto

I05 - Tirame un centro

I06 - Años bisiestos

I07 - Recibo de sueldo

I08 - Dibujando un triángulo rectángulo

I09 - Dibujando un triángulo equilátero

Bibliografía

- Price, M. J. (2020). *C#9 and .NET 5 - Modern Cross-Platform Development.* (5th ed., pp. 41-202). Packt Publishing.
- *.NET Core and .NET 5 Support Policy*. Microsoft Docs. https://dotnet.microsoft.com/platform/support/policy/dotnet-core.
- *.NET Architectural Components*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/components.
- *Common Language Runtime (CLR) overview*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/clr.
- *How is C# Compiled?*. Manning free content center. https://freecontent.manning.com/how-is-c-compiled/.
- *.NET Glossary*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/glossary.
- *Common Type System & Common Language Specification.*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/common-type-system.
- *Format types in .NET*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types
- *Console Class*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-5.0

*Last updated on **8/23/2021** by **mauriciocerizza***

Apuntes - Introducción a .NET y C#

Introducción a .NET

.NET (*pronunciado como "dot net"*) es una plataforma gratuita y de código abierto que nos provee una serie de herramientas y programas para construir fácilmente una gran variedad de software, así como el entorno necesario para ejecutarlo sobre distintos sistemas operativos y tipos de arquitectura.

Implementaciones de .NET

Actualmente existen cuatro implementaciones distintas de .NET las cuales conviven y tienen muchos puntos en común:

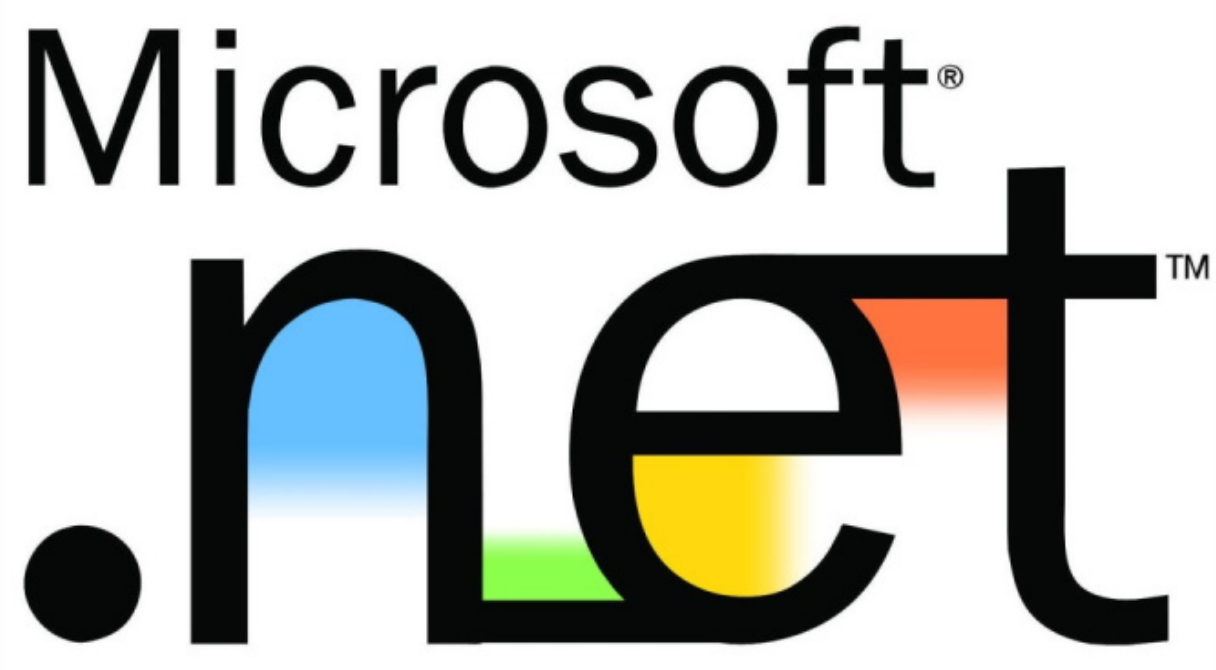
- .NET Framework
- .NET Core
- Xamarin
- .NET 5

> **INFORMACIÓN**
>
> Para quienes les interese profundizar la historia de .NET, Richard Campbell nos resume 20 años en menos de una hora: The History of .NET - Richard Campbell.

.NET Framework

.NET Framework es una plataforma de desarrollo que fue diseñada por Microsoft a fines de la década de los 90s y cuya primera versión fue lanzada en 2002. Desde entonces y hasta 2019 fue recibiendo múltiples actualizaciones. Su versión final es la 4.8.

Nació de la necesidad de unificar los distintos componentes necesarios para construir, implementar y ejecutar una aplicación desarrollada para Windows.



Los pilares sobre los que se construyó esta implementación son el **Common Language Runtime (CLR)**, un programa que se encarga de todo lo necesario para ejecutar una aplicación construida con .NET, y la **Base Class Library (BCL)**, una enorme biblioteca con funcionalidades útiles para construir software. Ambas herramientas son comunes a todos los lenguajes de programación soportados por la plataforma.

Así, .NET Framework permitió desarrollar software cuyo código fuente fuera agnóstico de la plataforma de destino. El único componente dependiente de la computadora sobre la que se ejecutaba era el entorno de ejecución encargado de ejecutar ese código. Llegó a soportar hasta 22 lenguajes de programación distintos.

Actualmente se sugiere utilizar esta implementación únicamente para mantener o extender software originalmente programado sobre la plataforma. No se recomienda su uso para nuevos proyectos.

Mono/Xamarin

A fines del año 2000 Microsoft publicó la **Common Language Infrastruture (CLI)**, especificando cómo construir código ejecutable y un entorno de ejecución que permitiera usar múltiples lenguajes de alto nivel en diferentes plataformas, eliminando así la necesidad de reescribir el código fuente para cada una de ellas. Todas las implementaciones de .NET cumplen con las especificaciones del Common Language Infrastructure.

Originalmente, Microsoft sólo ofrecía un entorno de ejecución para Windows, el Common Language Runtime. Miguel de Icaza de la empresa Ximian, se interesó en la plataforma y empezó a evaluar la posibilidad de desarrollar un entorno de ejecución para Linux. Así nace el proyecto de código abierto **Mono**, el cual vio la luz por primera vez en 2004.

Ximian fue adquirida por Novell en 2003. A su vez, Novell fue adquirida por Attachmate en 2011. Esto último fue sucedido por múltiples despidos dentro de la flota de trabajo que había sido heredada de Novell, incluyendo a Miguel de Icaza. A raíz de esto, Miguel y gran parte del equipo original de Mono, terminan fundando Xamarin. Xamarin fue la empresa que continuó soportando el desarrollo de Mono.

Para este tiempo, lograron adquirir los permisos y licencias para continuar trabajando sobre MonoTouch (para iOS) y Mono for Android, los cuales pasaron a llamarse respectivamente Xamarin.iOS y Xamarin.Android. En 2012 Xamarin lanza Xamarin.Mac, que permitía desarrollar aplicaciones para MacOS con C#. En 2013 lanzan el IDE Xamarin Studio, un cambio de marca del IDE original para Mono "Monodevelop" y la integración con Visual Studio, lo que permitió usar Visual Studio para crear aplicaciones para Android, iOS y Windows.

En 2016, la empresa es adquirida por Microsoft que liberó el código fuente del SDK de Xamarin y comenzó a ofrecer gratuitamente las integraciones y características para Visual Studio.

Fue así como Mono dio origen a la plataforma para desarrollo mobile **Xamarin** y otras como **Unity**, que se utiliza para el desarrollo de videojuegos multiplataforma. La versión final de Mono es la 5.0.

> **CURIOSIDAD**
>
> El nombre de Xamarin proviene de los monos tamarinos (tamarins, en inglés), sólo que se reemplaza la T por una X. Similar a como Ximian, la empresa original de Miguel de Icaza, deriva de simians (simios)*.

.NET Core

Cuando nació .NET Framework, el modelo de negocio de Microsoft consistía principalmente en vender licencias de Windows y Office. No es extraño entonces que se tratara de una implementación ligada tan estrechamente a Windows. Hoy vivimos en un mundo multi-plataforma, donde el desarrollo para dispositivos móviles y la infraestructura en "la nube" transformaron a Windows en un sistema operativo mucho menos relevante. Microsoft se adaptó al cambio y su modelo de negocio pasó a ser vender sus servicios en la nube.

Fue bajo este nuevo contexto que en 2016 se lanza la primera versión de **.NET Core**, una nueva implementación multi-plataforma y de código abierto. Para esta implementación se rescribió desde cero toda la plataforma de .NET, logrando mejoras de rendimiento, eficiencia y un enfoque de desarrollo mucho más moderno.

.NET Core trajo un nuevo entorno de ejecución ***multi-plataforma*** llamado **CoreCLR** con soporte para Windows, Linux y MacOS. También incluyó una nueva Base Class Library más liviana y simple, conocida como **CoreFX**.

Con esta implementación también se eliminaron grandes piezas de código que estaban estrechamente ligadas a tecnologías legado y exclusivas de Windows. Muchas fueron modularizadas en forma de **paquetes NuGet**, bibliotecas externas que uno puede incorporar a sus proyectos a necesidad. ***Modularizar*** .NET para que tenga menos depedencias hizo de .NET Core una implementación mucho más ligera y simple que .NET Framework, a su vez permitiendo que las nuevas actualizaciones tengan menos impacto sobre la plataforma.

La versión final de .NET Core es la 3.1.

.NET Standard

Para 2019 existían y convivían tres implementaciones de .NET:

- .NET Core para multi-plataforma y nuevos desarrollos.
- .NET Framework para aplicaciones legado.
- Xamarin para aplicaciones mobile.

Si bien cada una de estas estaba especializada en escenarios diferentes, condujo al problema de que un desarrollador tenía que aprender a trabajar con las tres plataformas.

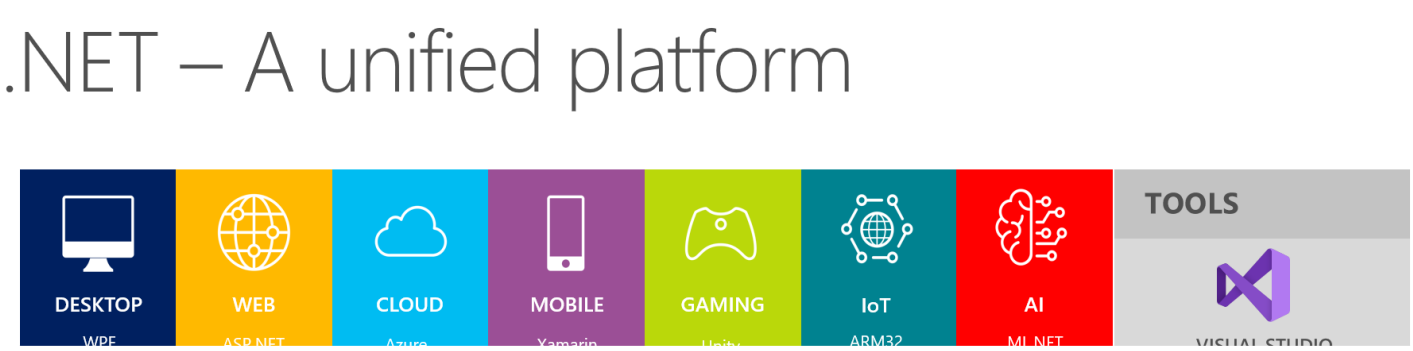
Esto condujo a Microsoft a publicar **.NET Standard**, una especificación que debe cumplir e implementar la Base Class Library de cualquier plataforma .NET. Esto trajo uniformidad y permitió a los desarrolladores compartir un mismo código fuente entre las distintas implementaciones de la plataforma.

.NET 5 y versiones superiores

En 2020 .NET Core cambia de nombre a .NET y se lanza **.NET 5**, la primera versión unificada de la plataforma con soporte para construir aplicaciones de todo tipo (cloud, desktop, mobile, gaming, y más). Esta nueva implementación combina las herramientas y frameworks de .NET Framework, .NET Core y Mono/Xamarin.

Al haber una sola plataforma unificada de .NET, la necesidad de .NET Standard (que dio la base para soñar con la posibilidad .NET 5) se redujo significativamente.

Esta implementación reemplazará a todas las anteriores, convirtiéndose en un único estándar.

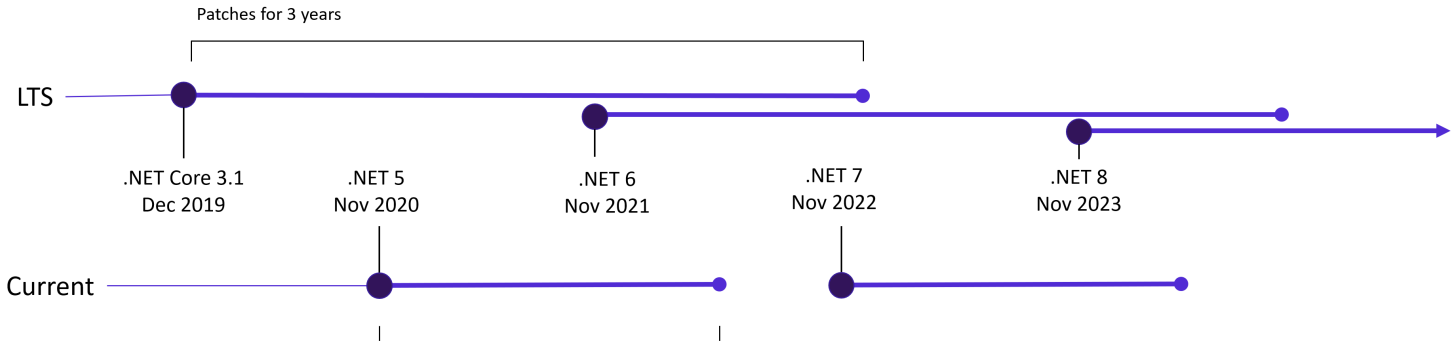


Niveles de soporte

Las versiones de .NET se categorizan en dos grandes grupos que determinan por cuántos años tendrán soporte (mantenimiento):

- **Long-Term Support (LTS)**: Son versiones estables y que se actualizarán con poca frecuencia. Se garantiza su soporte por 3 años desde su lanzamiento. Por ejemplo, la versión 3.1 que fue lanzada en Diciembre de 2019 tiene soporte hasta Diciembre de 2022.
- **Current**: Contienen las últimas mejoras y tienden a actualizarse con frecuencia. Reciben actualizaciones hasta 18 meses después de su lanzamiento. Por ejemplo, la versión 5.0 que fue lanzada en Noviembre de 2020 tiene soporte hasta Mayo de 2022.

Ambas reciben correcciones críticas relacionadas a temas de seguridad. Debemos mantenernos al día con estas actualizaciones para obtener soporte. Por ejemplo, si tenemos instalada la versión 5.0.0 y existe una versión 5.0.1, necesitaremos estar actualizados a la 5.0.1 para obtener soporte.



Características de .NET

Multi-plataforma

Existió una época donde esta plataforma sólo nos permitía trabajar para Windows, pero esos tiempos quedaron muy atrás. Desde la salida de .NET Core en 2016, podemos implementar nuestros sitios web, aplicaciones para servidores y programas de consola también en Linux y MacOS.

Open Source

El código fuente de .NET es público y es mantenido por miles de desarrolladores y compañias. Es soportado por .NET foundation, una organización sin fines de lucro, la cual se encarga de promover el desarrollo y la colaboración alrededor del ecosistema de .NET.

Multi-lenguaje

.NET soporta varios lenguajes de programación, los cuales se pueden utilizar para programar sobre la plataforma:

C# → Lenguaje orientado a objetos con una sintaxis similar a C y JAVA.

F# → Lenguaje orientado principalmente a la programación funcional, de sintaxis liviana.

Visual Basic → La sintaxis de este lenguaje es la que más se asemeja al lenguaje humano (inglés), lo que facilita el trabajo para personas sin experiencia en el desarrollo de software.

Componentes de .NET

Todas las implementaciones de .NET incluyen los siguientes componentes:

- Uno o más entornos de ejecución.
- Una biblioteca de clases base.
- Infraestructura y componentes comunes.
- Opcionalmente, uno a más frameworks para desarrollo de aplicaciones.
- Opcionalmente, herramientas de desarrollo adicionales.

Common Language Runtime

Un **entorno de ejecución (*runtime*)** es un programa encargado de administrar la ejecución de un programa. El runtime de .NET Framework y .NET 5+ se conoce como **Common Language Runtime (CLR)**.

Entre sus tareas se encuentra:

- Administrar el uso, asignación y liberación de memoria.
- Genera y compila código para que el programa pueda ejecutarse sobre una máquina en concreto.
- Sirve como capa de abstracción del hardware, permitiendo a los desarrolladores programar de forma agnóstica a la plataforma de destino.
- Manejo de errores en tiempo de ejecución.
- Provee funcionalidades para lenguajes orientados a objetos.
- Soporte para programación multi-hilo.
- Cuestiones de seguridad y rendimiento.

Es probable que no reconozcas algunos términos de la lista de tareas del CLR, pero iremos profundizando sobre esto durante la cursada. Por ahora entendamos que es un software que permite la

ejecución de las aplicaciones construidas con .NET y que nos genera una capa de separación entre el código fuente y el hardware donde se terminará ejecutando.

El runtime de .NET 5 y versiones superiores es el ***.NET 5 Common Language Runtime (.NET 5 CLR)***.

Base Class Library

La **base class library (BCL)** es una serie de bibliotecas (*libraries*) con funcionalidades de uso general que forman parte de los espacios de nombre *Microsoft* y *System*. Funciona como un framework de bajo nivel sobre el que están construidos los frameworks de aplicación de alto nivel que forman parte del ecosistema de .NET.

La biblioteca de clases de .NET 5 y versiones superiores es la ***.NET 5 Base Class Library***.

Frameworks

Un **marco de trabajo (*framework*)** define una forma de trabajo y nos brinda un conjunto de herramientas de alto nivel para desarrollar un tipo de aplicación en concreto.

Algunos ejemplos de frameworks para .NET son:

- **Windows Forms**. Será el framework que utilizaremos en clase para desarrollar aplicaciones de escritorio (*desktop*).
- **ASP.NET Core**. Es un framework para trabajar con desarrollo web, frontend y backend.
- **Windows Presentation Foundation (WPF)**. Se trata de otro framework para desarrollar aplicaciones de escritorio.

Frameworks vs bibliotecas

No se debe confundir a los marcos de trabajo con las **bibliotecas (*libraries*)**. Entre las diferencias más importantes encontramos:

Bibliotecas (Libraries)	**Marcos de trabajo (Frameworks)**
Se trata de una serie de funcionalidades para realizar operaciones específicas, bien definidas.	Definen una forma de trabajo y nos brinda un conjunto de herramientas de alto nivel que permiten desarrollar ciertos tipos de aplicaciones con facilidad.
Se componen de una colección de funciones y objetos auxiliares.	Se componen de múltiples bibliotecas y otras herramientas.

Bibliotecas (Libraries)	Marcos de trabajo (Frameworks)
Nosotros invocamos las funciones de la biblioteca a necesidad.	El framework invoca al código y maneja el flujo de ejecución.
El desarrollador tiene libertad y control para usar la biblioteca como desee.	El framework tiene un comportamiento por defecto y define un estándar para el desarrollo.

Herramientas e infraestructura

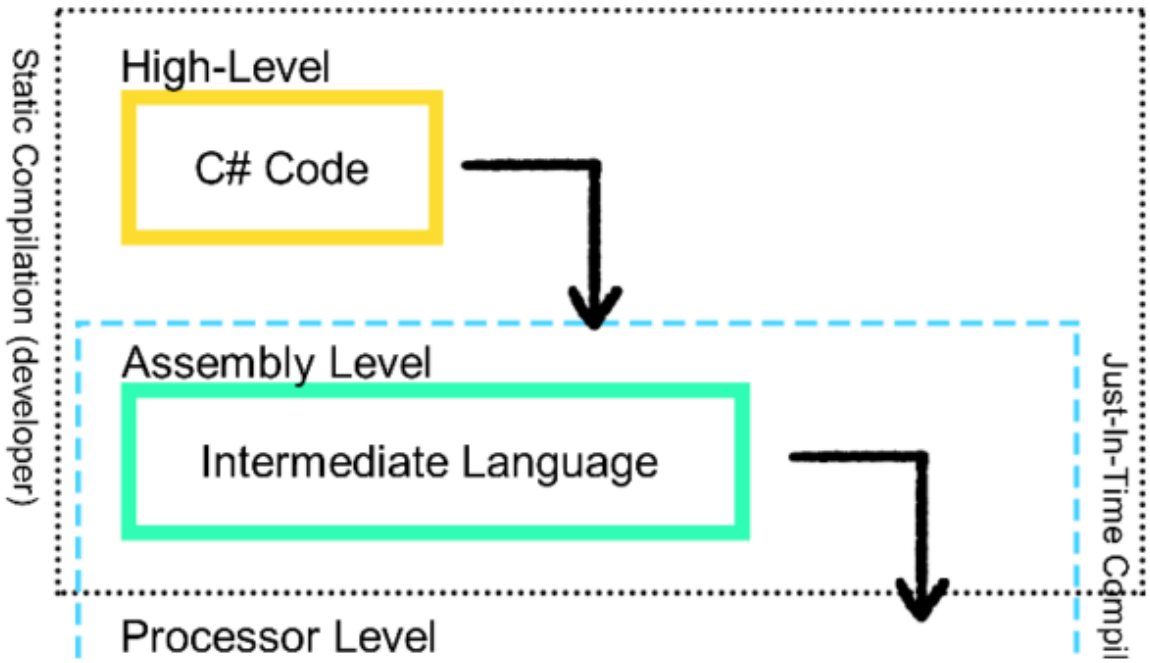
.NET integra las siguientes herramientas e infraestructura común:

- **Lenguajes y sus compiladores**.
- **.NET CLI** → Una interfaz de línea de comandos que nos provee una serie de instrucciones de consola que nos permitirán desarrollar, construir, ejecutar y públicar aplicaciones construidas con .NET.
- **MSBuild** → Un motor para cargar y construir nuestras aplicaciones.
- **NuGet** → Un administrador de paquetes desde donde podremos incorporar distintas librerías a nuestros proyectos, muchas de ellas desarrolladas por la comunidad.

Proceso de compilación

Todos los lenguajes de .NET siguen el mismo proceso de compilación.

El proceso de compilación pasa por tres estados y se divide en dos etapas.



Etapa 1: Compilación estática

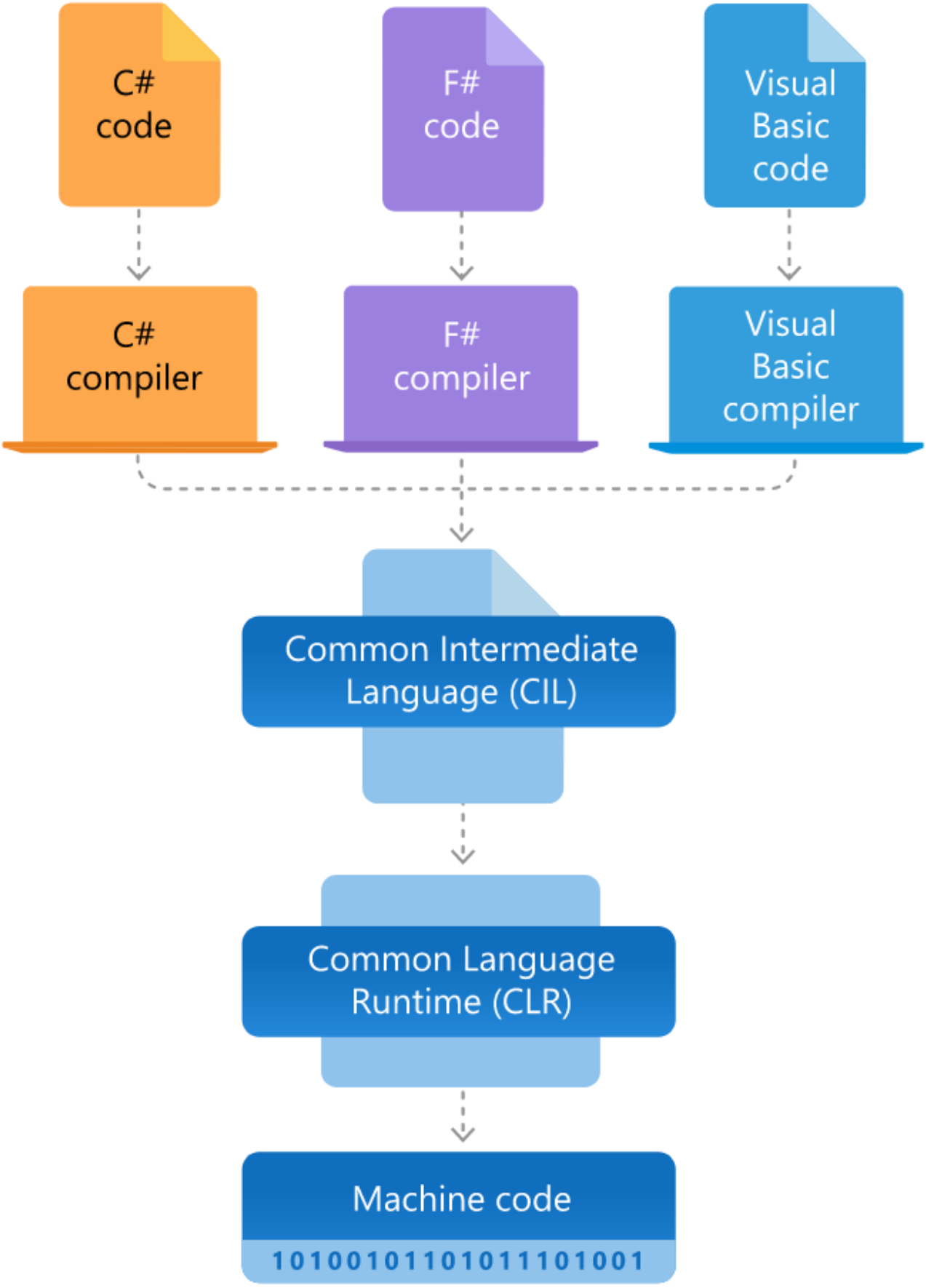
El **código fuente (*source code*)** de los programas construidos sobre .NET puede estar escrito con cualquiera de los lenguajes que soporta la plataforma, entre los que se encuentran C#, F# y VB.NET. Cada lenguaje tiene su propio compilador, el de C# se conoce como *Roslyn*. El **compilador** se encarga de traducir el código fuente a un lenguaje ensamblador conocido como **lenguaje intermedio (*intermediate language*)**.

El compilador se ejecuta cuando un desarrollador o proceso automatizado dispara el proceso de construcción (*build*).

El resultado de la compilación serán archivos que contendrán el lenguaje intermedio y serán aquellos que se distribuirán a los usuarios finales. En Windows estos archivos llevan la extensión *.exe* para ejecutables y *.dll* para bibliotecas.

Etapa 2: Compilación just-in-time

Cuando ejecutamos un programa de .NET, el sistema operativo invoca al Common Language Runtime. Luego, el CLR compila el lenguaje intermedio a **lenguaje nativo (máquina)** en un proceso que se conoce como **compilación just-in-time (JIT)**. Este tipo de compilación tiene como beneficio que abstrae al código fuente de la máquina y el sistema operativo sobre el que se terminará ejecutando el programa, el dependiente es el compilador JIT y el entorno de ejecución. Como desventaja, la compilación JIT realentiza la ejecución del programa ya que se debe esperar a que se compile el lenguaje intermedio.



Tiempo de compilación

Se denomina **tiempo de compilación (*compile-time*)** al intervalo de tiempo en el que un compilador compila código escrito en un lenguaje de programación a una forma de código ejecutable por una

máquina.

El compilador normalmente realiza un chequeo de sintaxis y una optimización del código generado.

El tiempo de compilación no sucede en los lenguajes interpretados debido a que estos no necesitan compilarse.

Tiempo de ejecución

Se denomina **tiempo de ejecución (*runtime*)** al intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo. Este tiempo se inicia con la puesta en memoria principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones. El intervalo finaliza en el momento en que este envía al sistema operativo la señal de terminación, sea esta una terminación normal, en que el programa tuvo la posibilidad de concluir sus instrucciones satisfactoriamente, o una terminación anormal, en el que el programa produjo algún error y el sistema debió forzar su finalización.

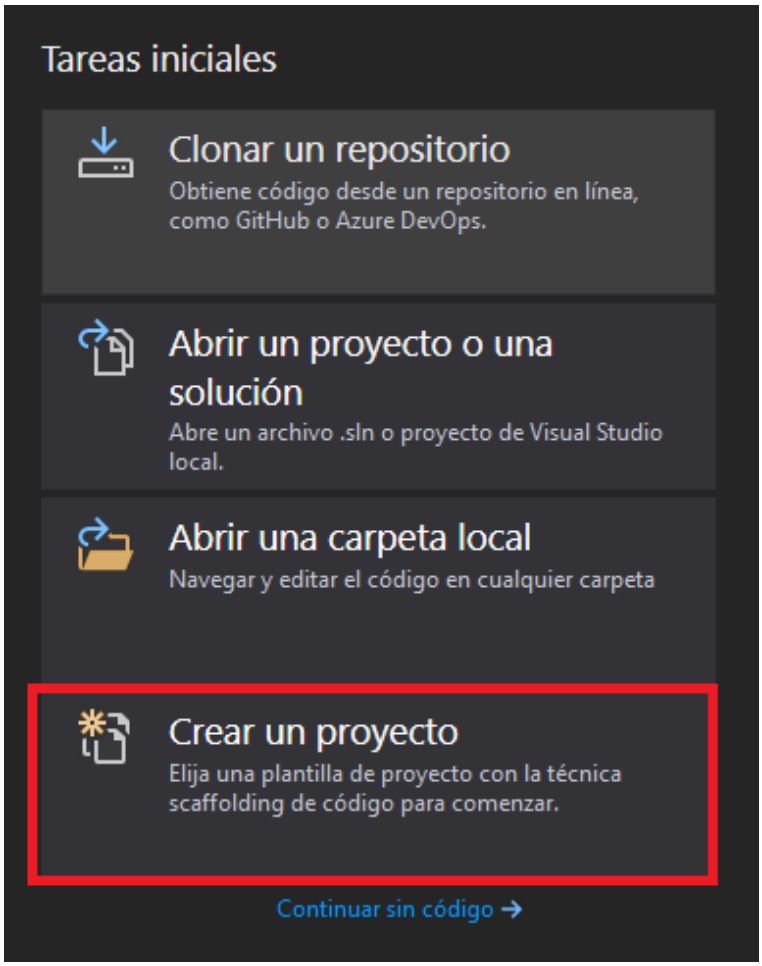
Este término suele emplearse en oposición a tiempo de compilación, para indicar si una acción o hecho sucede en uno u otro tiempo.

*Last updated on **8/23/2021** by **mauriciocerizza***

Crear un proyecto de consola

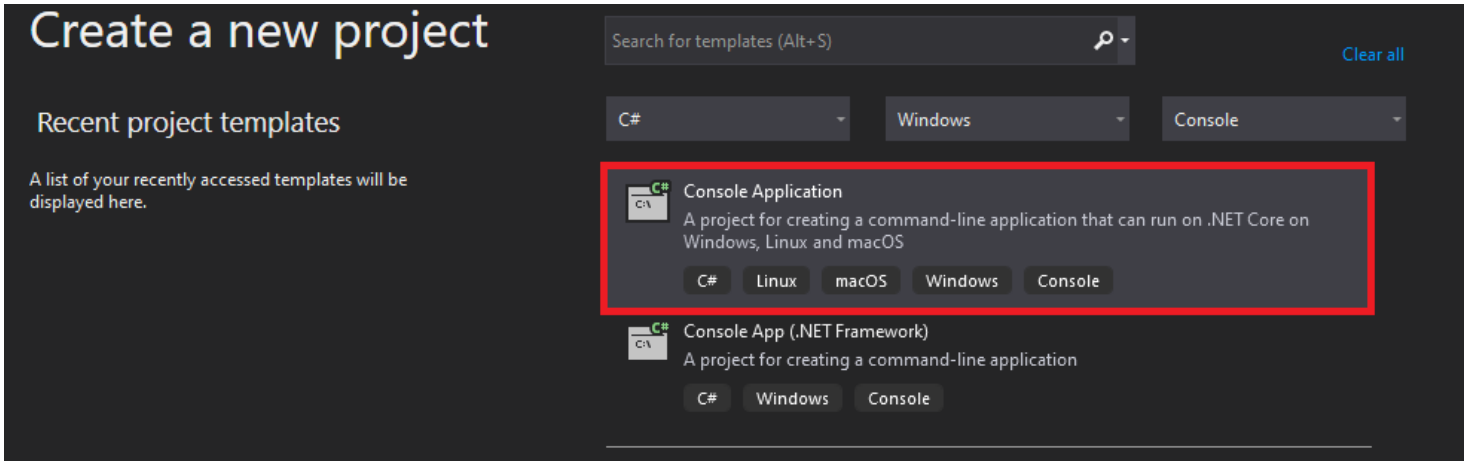
Exploremos el entorno de trabajo y pongamos en ejecución nuestra primera aplicación de consola.

Lo primero que tendremos que hacer es abrir `Visual Studio`, nos mostrará la siguiente pantalla con acciones rápidas para arrancar:



Presionaremos en `Crear un proyecto`. Lo siguiente que encontraremos es una serie de plantillas o templates de los distintos tipos de proyectos que podemos encarar con la plataforma. Arriba tendremos una barra de búsqueda y filtros por lenguaje, plataforma y tipo de aplicación.

Elegiremos `Aplicación de consola`.



Lo siguiente será elegir un nombre para el proyecto y otro para la solución (o el mismo) y presionar el botón "Crear".

Un **proyecto** es una estructura que nos permitirá construir nuestros programas con .NET y compilarlos como una unidad independiente, mientras que a una **solución** la podemos ver como una agrupadora de proyectos. Normalmente un programa de .NET está compuesto por varios proyectos inter-relacionados dentro de una solución.

Una vez creado nos aparecerá el siguiente código:

```
using System;

namespace Hola_Mundo
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Este es el código mínimo sobre el cual arrancaremos todas nuestras aplicaciones de consola. Nos adentraremos en los detalles más adelante.

Para ejecutar este programa tenemos 2 opciones, o presionar la tecla `F5` o el botón de "play" en la barra de acciones de arriba.

Seguido veremos la consola con la salida en texto "*Hello World!*".

¡Felicitaciones, acabás de crear tu primer programa con C#!

> **TIP**
>
> Te recomendamos crear una nueva solución por cada ejercicio que vayas a realizar. También mantener todas las soluciones en una única carpeta que esté versionada con Git y enlazada a tu repositorio en GitHub.

*Last updated on **8/23/2021** by **mauriciocerizza***

Introducción a C#

Durante nuestro viaje a través de las características comunes de los lenguajes de alto nivel y la programación orientada a objetos nos acompañará el lenguaje de programación C#. **C#** (pronunciado 'si sharp' en inglés) es un lenguaje de programación diseñado para generar programas sobre la plataforma .NET.

Tal vez algunas de las características de este lenguaje les parezcan similares a otros lenguajes que conozcan, y están en lo correcto. El equipo de C#, desde sus inicios, no dudó en tomar grandes ideas de otros lenguajes y reformarlas para incorporarlas a C#. Las principales influencias han sido Java (sobre todo en los inicios), y más cerca en el tiempo el mismísimo F#. También podemos encontrar similitudes con C, C++ y JavaScript.

Fue diseñado por el ingeniero de Microsoft Anders Hejlsberg, quien también está involucrado en el desarrollo de Typescript desde 2012.

Características de C#

Compilación híbrida

Para construir programas con C# se requiere de un **compilador** para traducir todo el código fuente a un lenguaje que permita ejecutar la aplicación en una computadora. C# se compila primero a un lenguaje intermedio y posteriormente al ejecutarse es recompilado a lenguaje nativo/máquina.

Esto se contrapone a los lenguajes cuyo código fuente es ejecutado línea a línea y en tiempo de ejecución por un **intérprete**.

Orientado a objetos

El lenguaje ofrece una serie de características orientadas a objetos como soporte para herencia, polimorfismo y encapsulamiento. El **paradigma de programación orientada a objetos** se enfoca en en las relaciones entre clases y objetos. Profundizaremos este tema durante la cursada.

Orientado a componentes

También presenta características que permiten el **desarrollo basado en componentes**. Un componente de software individual es un paquete de software o un módulo que encapsula un conjunto de datos o funciones relacionadas. Se trata de construir piezas de software débilmente acopladas (poco dependientes entre si), permitiendo la reutilización e intercambio de las mismas.

Seguridad de tipos

Es un lenguaje principalmente de tipado estático.

Recordemos la diferencia entre tipado estático y tipado dinámico.

Tipado estático → Decimos que un lenguaje es de tipado estático cuando los tipos de las variables tienen que definirse antes de compilar el programa. **Tipado dinámico** → Decimos que un lenguaje es de tipado dinámico cuando los tipos de las variables se definen durante la ejecución del programa.

C# cuenta con características que permiten trabajar con tipado dinámico, pero no es lo más común.

Garbage Collection

Está integrado un programa especial que se encarga de la liberación de memoria no utilizada en el segmento heap. A este programa se lo conoce como **Garbage Collector** y nos evita tener que programar explícitamente instrucciones para la administración de memoria. Esto facilita el desarrollo y genera beneficios de seguridad y eficiencia.

Sistema de tipos unificado

Todos los tipos de datos en C#, incluyendo los primitivos (como `int` o `double`), heredan de la clase `System.Object`. Esto significa que todos los tipos de dato tienen una serie de operaciones/funcionalidades en común. Los valores de cualquier tipo pueden ser almacenados, transportados y operadores de una manera consistente.

Case sensitive

C# es un lenguaje que distingue mayúsculas de minúsculas. No es lo mismo una variable llamada *numero* a una llamada *Numero*.

Gramática de C#

Los lenguajes de computación, al igual que los lenguajes humanos, tienen sintaxis y semántica.

La **sintaxis** es una serie de reglas que define las combinaciones correctas de símbolos y el orden para formar sentencias y expresiones válidas. En otras palabras, define cómo debe estar escrito y estructurado un lenguaje para construir sentencias válidas.

Algunos ejemplos de sintaxis de C# son terminar las sentencias con un punto y coma, o encerrar las expresiones condicionales de un bloque if dentro de paréntesis.

Si la sintaxis es inválida el programa no compila.

El **vocabulario** del lenguaje son una serie de palabras reservadas y operadores que sirven para construir sentencias y expresiones siguiendo las reglas de sintaxis.

La **semántica** es el signicado que surge de la combinación de esas sentencias y expresiones con una sintaxis válida. ¿Qué instrucción generan a la computadora? ¿tiene sentido lógico?.

Por ejemplo, en español la oración "*Él tomar agua*" tiene una sintaxis incorrecta pero una semántica que se puede inferir. Por otro lado, en la oración "*Él toma arroz*" la sintaxis es correcta pero no la semántica, el significado no es coherente.

Analicemos el siguiente fragmento de código.

```
if (condicion)
{
    Console.WriteLine("Entiendo sintaxis y semántica.");
}
else
{
    Console.WriteLine("No entiendo nada.");
}
```

Entre los elementos de sintaxis podemos identificar empezar el bloque con la palabra reservada `if` seguida de la expresión condicional entre paréntesis, el uso de llaves y la terminación de la sentencia `Console.WriteLine` con `;`. Si pusieramos el if sin paréntesis o nos faltara una de las llaves o nos olvidáramos de poner el punto y coma, el programa no compilaría ya que la sintaxis es inválida.

De la semántica se desprende que si la expresión condicional se cumple (es verdadera) entonces se mostrará en la salida de la consola la frase "*Entiendo sintaxis y semántica*", de lo contrario mostrará "*No entiendo nada*".

Sintaxis de C#

Sentencias

Cuando escribimos en español, marcamos el fin de una oración con un punto. Una oración se compone de múltiples palabras y frases con un orden específico. Los lenguajes de programación también tienen reglas de sintaxis.

C# indica el fin de una declaración con un punto y coma. Una **declaración (*statement*)** puede estar compuesta de múltiples **variables** y **expresiones (*expressions*)**. En el siguiente ejemplo, `sueldoNeto` es una variable y `sueldoBruto - aportes - impuestos` es una expresión compuesta de 3 **operandos** (`sueldoBruto`, `aportes` y `impuestos`) y los **operadores** `-`.

```
decimal sueldoNeto = sueldoBruto - aportes - impuestos;
```

El orden de los operadores y los operandos importa, si los cambiaramos de lugar daría un resultado completamente distinto.

Comentarios

Los comentarios le indican al compilador que ignore su contenido. Permite realizar aclaraciones o deshabilitar código temporalmente.

Para realizar un comentario en C# utilizamos la doble barra `//`. Lo que siga a la doble barra será ignorado por el compilador hasta el fin de la línea.

```
// Falta descontar los aportes.
decimal sueldoNeto = sueldoBruto - impuestos;
```

Si queremos escribir un comentario multi-línea se utiliza `/*` al inicio y `*/` al final, todo el contenido entre esos dos símbolos será ignorado por el compilador.

```
/*
    Estamos en programación II de UTN Fra.

    string saludo = "¡Hola Mundo!"

    Aprendemos a realizar comentarios en C#.
*/
```

> **INFORMACIÓN**
>
> El atajo de teclado de Visual Studio para realizar comentarios es `CTRL + K + C`, y para deshacerlos `CTRL + K + U`.

Bloques

En C# para escribir un bloque de código se utilizan llaves `{}`. Si hacemos una comparación con el lenguaje humano, lo bloques serían análogos a los párrafos. Los bloques empiezan con una declaración que define el contenido del bloque.

En el siguiente ejemplo vemos que tenemos 3 bloques anidados. Uno definido por la palabra clave `namespace`, otro por `class` y el último está definido por la firma del *método* (similar a un función).

```
using System; // El ; indica el fin de la declaración

// 'namespace' define el tipo de bloque. 'Hola_Mundo' es un nombre o identificador
para ese bloque.
namespace Hola_Mundo
{ // La llave de apertura { indica el inicio de un bloque.
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!"); // Otra declaración.
        }
    }
} // La llave de cierre } indica el fin de un bloque.
```

Vocabulario de C#

El vocabulario de C# está compuesto de **palabras clave (*keywords*)**, **símbolos** y **tipos**.

Entre las palabras claves encontraremos `using`, `namespace`, `class`, `static`, `int`, `string`, `double`, `bool`, `if`, `switch`, `break`, `while`, `do`, `for`, `foreach` y más.

Entre los símbolos trabajaremos con `"`, `'`, `+`, `-`, `*`, `/`, `%`, `@`, `$`,entre otros.

Métodos, los verbos de C#

Los verbos en español representan acciones como *correr*, *jugar*, *saltar*. De la misma forma, los **métodos** en C# representan acciones que podrán ser ejecutadas por alguna parte del programa. Son similares a las funciones en otros lenguajes.

```
using System;

namespace Hola_Mundo
{
    class Program
    {
        // Acá comienza la declaración del método Main.
        static void Main(string[] args)
        {
            /* Acá se invoca/llama al método WriteLine
            y se le pasa como argumento (dato o valor de entrada) un string.
            */
            Console.WriteLine("Hello World!");
        }
        // Acá finaliza la declaración del método Main.
```

```
    }
}
```

Nombres de métodos

Los nombres de los métodos deben ser *verbos* escritos con **notación Pascal (*Pascal Case*)**. La notación Pascal, también conocida como Upper Camel Case, indica que la primera letra de cada palabra dentro de un nombre compuesto debe estar en maýuscula.

Ejemplos: *MostrarDatos() ConfirmarCompra() RechazarPedido() CancelarOperacion()*

Método Main

El método `Main` es el **punto de entrada (entry point)** de todos los programas en C#, es decir, el primer método en ejecutarse.

Analicemos su firma:

- `static`: Es un modificador que permite ejecutar un método sin tener que instanciar a una variable (sin crear un objeto). El método `Main` debe ser estático.
- `void`: Indica el tipo retorno del método. `void` se utiliza cuando el método no retorna ningún valor. No necesariamente tiene que ser void, se puede especificar otro tipo de retorno.
- `string[] args`: Es un array de tipo `string`. Si la aplicación de consola recibe argumentos para su ejecución, los valores estarán dentro de este array. Este parámetro es opcional.

Tipos, atributos y variables. Los sustantivos de C#.

En español los sustantivos son palabras que se utilizan para denominar seres, entidades u objetos.

Por ejemplo, *Federico* es el nombre de un profesor. La palabra "profesor" nos indica el papel que juega *Fede* dentro del contexto de la universidad. Podemos decir que *Fede* es de tipo `Profesor`. `Profesor` no es el único tipo dentro del contexto de la facultad, también están `Alumno`, `Secretario`, `Coordinador`, `LibretaUniversitaria`, `Materia`, `Aula`, `Nota`, entre otros. Los **tipos** son sustantivos que categorizan cosas.

`legajoDocente` y `antiguedad` son características que forman parte de lo que describe a *Fede* como profesor, son **atributos**.

Por otra parte, *Fede* no es el único ser de tipo `Profesor`, también están *Lucas*, *Ezequiel* y *Mauricio*. Cada uno de estos seres o manifestaciones concretas de un tipo son **variables**, sustantivos que se refieren a una cosa específica.

Trabajando con variables

Todas las aplicaciones procesan datos de algún tipo. El origen de los datos puede ser variado y pueden ser almacenados temporalmente en la memoria asignada al programa en ejecución. Cuando el programa finaliza se pierden los datos en memoria.

Las variables se utilizan para almacenar los datos en memoria y así poder procesarlos.

En C# debemos definir un tipo de dato apropiado a la hora de declarar variables. El tipo de dato definirá un conjunto de métodos y atributos para trabajar con ese valor, así como el tamaño que podrá ocupar el valor en la memoria.

Las variables locales, es decir aquellas declaradas dentro de métodos, existen sólo durante la ejecución de ese método. Si se trata de tipos de valor, la memoria es liberada inmediatamente al abandonar el método. Si se trata de tipos de referencia dependerán del proceso de *garbage collection*.

Para declarar una variable se debe colocar el tipo de dato seguido de un identificador (nombre):

```
string nombre;
int horasAsignadas;
```

Operadores de asignación

Para asignar un valor a la variable se utiliza el operador de asignación `=`. Lo que está a la derecha del operador se lee y almacena en la variable de la izquierda.

```
string nombre = "Lautaro";
int horasTrabajadas = 8;
int precioPorHora = 100;
int costoTrabajo = horasTrabajadas * precioPorHora;

Console.WriteLine("{0} trabajó {1} horas. El costo por el trabajo es de ${2}.",
nombre, horasTrabajadas, costoTrabajo);
```

La salida del código anterior es:

```
Lautaro trabajó 8 horas. El costo por el trabajo es de $800.
```

Constantes

Las **constantes** representan valores *inmutables*, es decir que sus valores se definen en tiempo de compilación y no cambian durante el resto de la vida del programa.

Se declaran antecediendo el modificador `const` al tipo de dato y al identificador de la constante.

```
const decimal IVA = 0.21M;

decimal precioBruto = 100M;
decimal precioNeto = precioBruto + precioBruto * IVA;

Console.WriteLine("El precio bruto es {0} y aplicando IVA queda en {1}.",
precioBruto, precioNeto);
```

La salida del código anterior es:

```
El precio bruto es 100 y aplicando IVA queda en 121,00.
```

Nombres de variables y atributos

Los nombres de las variables y atributos deben ser *sustantivos* escritos con **notación Camel (*Camel Case*)**. La notación Camel, también conocida como Lower Camel Case, indica que la primera letra de un nombre va en minúscula y luego cada palabra adicional debe empezar en mayúscula.

Ejemplos: *edad resultado valorMaximo cotizacionUsdArs materiasAsignadas numerosPrimos*

Operador nameof

El operador `nameof()` retorna el nombre de una variable, tipo o atributo como una cadena de texto.

```
string nombre = "Lautaro";

Console.WriteLine("La variable {0} contiene el valor {1}.", nameof(nombre), nombre);
```

La salida del código anterior es:

```
La variable nombre contiene el valor Lautaro.
```

*Last updated on **8/23/2021** by **mauriciocerizza***

On this page

Common Type System

C# por defecto sólo trae algunas palabras y, estrictamente, no define ningún tipo. Todos los tipos que usamos en C# son provistos por el entorno de .NET. Durante la cursada aprenderemos sobre muchos de los tipos que tenemos disponibles en la plataforma y también crearemos nuevos.

El **Common Type System (CTS)** define un conjunto de tipos de datos común a todos los lenguajes soportados por .NET.

- Establece un marco de herramientas que habilita la ejecución de los distintos lenguajes de la plataforma.
- Provee un modelo orientado a objetos.
- Define un conjunto de reglas que todos los lenguajes deben seguir en lo que refiere a tipos.
- Provee una biblioteca que contiene los tipos primitivos básicos (Boolean, Int32, Byte, Char, etc).
- Define tipos de dato en dos categorías: de valor y de referencia.

Tipos de valor y tipos de referencia

Existen dos segmentos o categorías de memoria: la **pila (*stack memory*)** y el **montón (*heap memory*)**. La memoria stack es más rápiida pero limitada en tamaño. La memoria heap es más lenta pero más abundante.

> **STACK OVERFLOW**
>
> La famosa excepción Stack Overflow se lanza al llenarse el espacio de la memoria stack, que es muy limitado. Suele suceder cuando se producen llamadas recursivas accidentales o nos encontramos dentro de un loop infinito.

Los **tipos de valor (*value types*)** son tipos de dato representados por su valor real. Si son asignados a una variable, esa variable obtendrá una nueva copia del valor. **Todos los tipos de valor se almacenan en la pila.**

Los **tipos de referencia (*reference types*)**, al contrario, son tipos de dato representados por una referencia que apunta a un sector de memoria donde se encuentra el valor real. Si son asignados a una variable, esa variable almacenará la referencia y apuntará al valor original. No se realiza ninguna copia del valor. **Todos los tipos de referencia se almacenan en el montón.**

Categorías de tipos

.NET define cinco categorías de tipos de datos.

Categoría	Palabra clave	Valor/Referencia
Clases	***class***	Tipo de referencia
Estructuras	***struct***	Tipo de valor
Enumerados	***enum***	Tipo de valor
Interfaces	***interface***	Tipo de referencia
Delegados	***delegate***	Tipo de referencia

Entraremos en el detalle de cada una de estas categorías a lo largo de la cursada.

Aliases

Algunas de las palabras clave de C# como `double`, `int` o `string` son **alias (*aliases*)** que representan tipos proveidos por la implementación de la plataforma .NET donde se ejecuta C#. Por ejemplo, `int` es un alias para el tipo `System.Int32`.

Literales

Los **valores literales (*literal value*)** son una notación que representa un valor fijo. Dependiendo el tipo de dato, existe una notación diferente para los literales.

```
string saludo = "Hola Mundo"; // "Hola Mundo" es un literal de texto que se está asignando a la variable 'saludo' de tipo string.

int numero = 5; // 5 es un literal numérico que se está asignando a la variable 'numero' de tipo entero.
```

Caracteres

Cuando trabajamos con un caracter individual, como podría ser una letra, el tipo a utilizar es `char`.

Los literales de este tipo deben estar encerrados por comillas simples `'`.

Los `char` son tipos de valor.

```
char simbolo = '$';
char letra = 'Z';
char numero = '1'; // '1' se almacena como tipo char, no es numérico.
```

Strings

Cuando trabajamos con texto formado por múltiples caracteres el tipo a utilizar es `string`.

Los literales de este tipo deben estar encerrados por comillas dobles `"`.

Los `string` son tipos de referencia. Internamente son arrays de `char`, incluso pueden ser recorridos con un bucle `for` o `foreach`.

Los `string` se pueden concatenar usando el operador `+`.

```
string nombre = "Florencia";
string telefono = "(+54) 9 11-12345-15432";
string contacto = nombre + " tiene el número de teléfono " + telefono;
```

Tipos numéricos

Los números son datos con los que vamos a realizar alguna operación aritmética (como sumar o multiplicar). Un DNI, un número de teléfono, un legajo, NO son números.

Todos los tipos numéricos primitivos de C# son tipos de valor.

Operadores aritméticos

Los **operadores aritméticos** nos permiten realizar operaciones aritméticas sobre una o más variables numéricas.

Operador	Nombre	Ejemplo	Resultado
`+`	Suma	4 + 2	6

Operador	Nombre	Ejemplo	Resultado
`-`	Resta	4 - 2	2
`*`	Multiplicación	4 * 2	8
`/`	División	4 / 2	2
`%`	Módulo o resto	4 % 2	0
`++`	Incremento	4++	5
`--`	Decremento	4--	3

> **INFORMACIÓN**
>
> Para más información sobre los operadores aritméticos, visita la documentación oficial.

Operadores de asignación

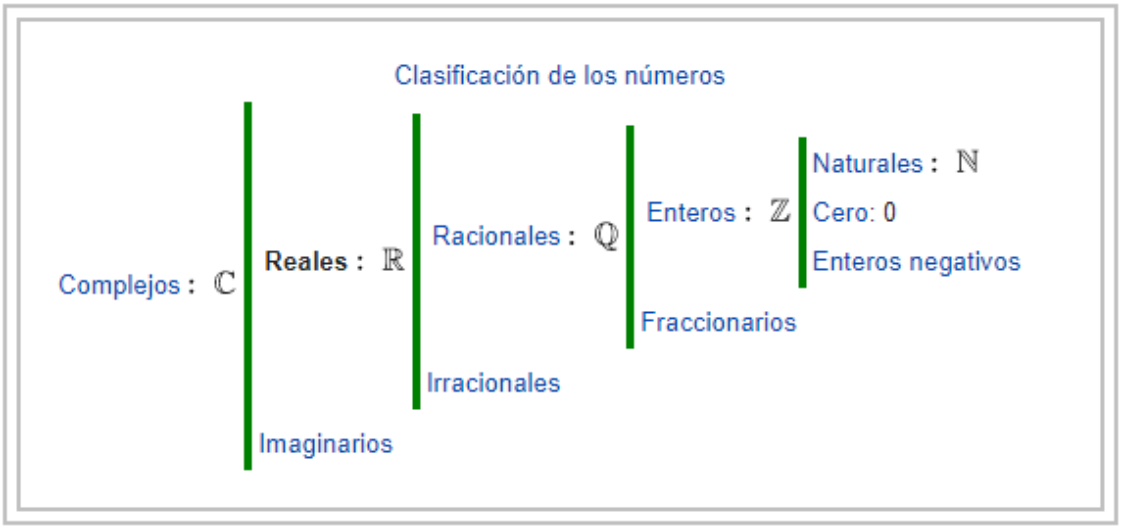
El operador de asignación se puede combinar con los operadores aritméticos:

```
int numero = 5;

numero += 2; // Es lo mismo que escribir numero = numero + 2;
numero -= 2; // Es lo mismo que escribir numero = numero - 2;
numero *= 2; // Es lo mismo que escribir numero = numero * 2;
numero /= 2; // Es lo mismo que escribir numero = numero / 2;
```

Enteros

El conjunto de **números reales** se puede dividir en dos subconjuntos: racionales e irracionales. Los **números racionales** son aquellos que pueden expresarse como la división de dos números enteros, los **números irracionales** son todos los demás. A su vez, el conjunto de los racionales se puede dividir en **números enteros** y **números fraccionarios**.



Los números enteros (sin punto decimal) se representan con los tipos `short`, `int` y `long`. Los enteros positivos incluyendo el cero se conocen como **números naturales** y se representan con los tipos enteros sin signo (***unsigned***) como `ushort`, `uint` y `ulong`.

Los literales de `long` deben ir acompañados del sufijo `L`. Para los enteros no es necesarario un sufijo.

```
uint numeroNatural = 10;

int numeroEntero = -5;

long numeroGrande = 20L;
```

Palabra clave	Rango	Tamaño	Tipo en .NET
`sbyte`	*-128* a *127*	Entero de 8-bit con signo	`System.SByte`
`byte`	*0* a *255*	Entero de 8-bit sin signo	`System.Byte`
`short`	*32.768* a *32.767*	Entero de 16-bit con signo	`System.Int16`
`ushort`	*0* a *65.535*	Entero de 16-bit sin signo	`System.UInt16`
`int`	*-2.147.483.648* a *2.147.483.647*	Entero de 32-bit con signo	`System.Int32`

Palabra clave	Rango	Tamaño	Tipo en .NET
`uint`	*0* a *4.294.967.295*	Entero de 32-bit sin signo	`System.UInt32`
`long`	*-9.223.372.036.854.775.808* a *9.223.372.036.854.775.807*	Entero de 64-bit con signo	`System.Int64`
`ulong`	*0* a *18.446.744.073.709.551.615*	Entero de 64-bit sin signo	`System.UInt64`

Punto flotante

Los tipos `float` y `double` almacenan números reales fraccionarios representados con notación de punto flotante de precisión simple y doble, respectivamente. En otras palabras, el tipo de dato `double` tiene una precisión dos veces mayor al tipo de dato float.

La representación de **punto flotante (*floating point*)** es una forma de notación científica usada en las computadoras con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas.

Decimales

Aunque el tipo de dato `double` es bastante preciso, existe un tipo de dato aún más preciso, que es el tipo de dato `decimal`. Es más preciso porque almacena los números como un gran entero y cambia el punto decimal. Por ejemplo `0.1` se almacena como `1` con una nota de cambiar el punto decimal un lugar a la izquierda, `12.75` se almacena como `1275` con una nota de cambiar el punto decimal dos lugares a la izquierda.

Entonces, si requerimos resultados precisos debemos usar `decimal`, sino debemos usar `float` y `double` que se procesan con mayor velocidad.

El comparar valores de punto flotante con el operador de igualdad puede dar lugar a errores.

```
double a = 0.1;
double b = 0.2;
double resultado = a + b;
bool resultadoEsperado = resultado == 0.3;

Console.WriteLine("La suma de {0} y {1} da {2}. ¿Resultado esperado? {3}", a, b,
```

```
resultado, resultadoEsperado);
```

La salida del código anterior es:

```
La suma de 0,1 y 0,2 da 0,30000000000000004. ¿Resultado esperado? False
```

En el ejemplo vemos que el resultado de la suma entre `0.1` y `0.2` cuando trabajamos `double` tiene un error de precisión y al comparar nos indica que no es igual a `0.3`.

```
decimal a = 0.1M;
decimal b = 0.2M;
decimal resultado = a + b;
bool resultadoEsperado = resultado == 0.3M;

Console.WriteLine("La suma de {0} y {1} da {2}. ¿Resultado esperado? {3}", a, b,
resultado, resultadoEsperado);
```

La salida del código anterior es:

```
La suma de 0,1 y 0,2 da 0,3. ¿Resultado esperado? True
```

Al trabajar con `decimal` no tenemos estos errores.

> **ADVERTENCIA**
>
> No se deben comparar valores `double` usando `==`. El tipo `double` no garantiza precisión porque existen algunos números que no pueden ser representados como valores de punto flotante.
>
> En 1991, durante la primera guerra del golfo, usar números de punto flotante le costó la vida a 28 soldados al no poder rastrear e interceptar un misil con precisión.

> **BUENA PRÁCTICA**
>
> Es una buena práctica usar `int` para números enteros y `double` para fraccionarios que no requieran precisión y que no serán comparados a otros valores. Usar `decimal` para dinero y valores donde la precisión es importante.

Notación de literales de punto flotante

Por defecto los literales fraccionarios (con punto decimal) son de tipo `double`. Los literales de `float` deben ir acompañados del sufijo `F`. Los literales de `decimal` deben ir acompañados del sufijo `M`.

```
float numeroFlotantePrecisionSimple = 2.5F;

double numeroFlotantePrecisionDoble = 2.5;

decimal numeroDecimal = 2.5M;
```

Notación binaria y hexadecimal

C# también permite escribir literales numéricos en binario (base 2) y hexadecimal (base 16). Un literal binario debe empezar con `0b`, mientras que uno hexadecimal con `0x`.

Desde C# 7.0 se puede usar el guión bajo `_` como separador de digitos con el único uso de mejorar la legibilidad del número.

En el siguiente ejemplo vemos como escribir el valor de dos millones en decimal, binario y hexadecimal.

```
// Notación decimal
int notacionBase10 = 2_000_000; // El _ es una ayuda visual para separar los dígitos. No cumple otra función.

// Notación binaria
int notacionBase2 = 0b_0001_1110_1000_0100_1000_0000;

// Notación hexadecimal
int notacionBase16 = 0x_001E_8480;
```

Si comparamos las variables con el operador de igualdad `==` observaremos que, al ser el mismo valor pero expresado con diferentes notaciones, la igualdad es verdadera.

Tipos booleanos

Los tipos booleanos sólo pueden contener dos valores literales `true` o `false`. Se usan principalmente en condiciones de bloques de selección

```
bool verdadero = true;
bool falso = false;
```

Operadores de igualdad

Los **operadores de igualdad** retornan un resultado booleano en base a si los operandos comparados son iguales o distintos.

`==` retornará `true` cuando la igualdad se cumpla, de lo contrario `false`. `!=` `true` cuando la igualdad NO se cumpla y `false` cuando los operandos sean iguales.

Operador	Nombre	Ejemplo	Resultado
`==`	Igualdad	"Programación II" == "Programación II"	true
`!=`	Desigualdad	"Programación II" != "Laboratorio de Computación II"	true

Los tipos de valor son iguales cuando sus valores son iguales. Los tipos de referencia son iguales cuando apuntan a la misma dirección de memoria.

Operadores de comparación

Los **operadores de comparación** también retornan un resultado booleano y sirven para comparar valores numéricos.

Retornarán `true` cuando la comparación se cumpla, de lo contrario `false`.

Operador	Nombre	Ejemplo	Resultado
`>`	Mayor que	4 > 4	false
`>=`	Mayor o igual a	4 >= 4	true
`<`	Menor que	4 < 2	false
`<=`	Menor o igual a	2 <= 4	true

Tipo object

Todos los tipos de datos en .NET derivan de un tipo de dato padre, la clase `System.Object`. `object` es un alias de `System.Object`.

Esto nos permite almacenar en una variable de tipo `object` cualquier valor.

```
object nombre = "Esteban";
object promedio = 9.99;

Console.WriteLine("{0} tiene un promedio de {1}.", nombre, promedio);
```

La salida del código anterior es:

```
Esteban tiene un promedio de 9,99.
```

Si quisiera acceder a la propiedad `Length` de la variable `nombre`, primero tendré que castear el valor a `string`.

```
object nombre = "Esteban";
int longitud = ((string) nombre).Length;

Console.WriteLine("{0} tiene {1} caracteres.", nombre, longitud);
```

La salida del código anterior es:

```
Esteban tiene 7 caracteres.
```

El operador de casteo `(tipo de dato)` le indica al compilador que el valor de una variable es en realidad de otro tipo. En el ejemplo le indicamos que el valor de `nombre` es de tipo `string`.

Una vez realizado el casteo podemos acceder a los atributos y métodos del tipo `string`.

> **ADVERTENCIA**
>
> No es una buena práctica usar el tipo `object`. C# es un lenguaje fuertemente tipado y se debe trabajar aprovechando las ventajas de definir tipos de datos concretos, usar `object` afecta la mantenibilidad del código y el rendimiento.

Tipo dynamic

`dynamic` es otro tipo especial que también puede almacenar cualquier valor. La diferencia con `object` radica en que nos permite utilizar los atributos y métodos del valor almacenado sin necesidad de un

casteo.

```
dynamic nombre = "Esteban";
dynamic promedio = 9.99;
int longitud = nombre.Length;

Console.WriteLine("{0} tiene {1} caracteres y un promedio de {2}.", nombre, longitud,
promedio);
```

La salida del código anterior es:

```
Esteban tiene 7 caracteres y un promedio de 9,99.
```

Estos tipos no se validan en tiempo de compilación, sino que lo hará el CLR durante en tiempo de ejecución. Por este motivo el *IntelliSense* de Visual Studio puede no funcionar para estas variables.

> **ADVERTENCIA**
>
> No se debe usar el tipo `dynamic` salvo que surja una necesidad concreta. El uso de este tipo afecta la mantenibilidad del código y el rendimiento.

Inferencia de tipos

El modificador `var` sirve para declarar variables para las cuales el tipo de dato será determinado por el valor que se le está asignando. Es una característica que nos quita la necesidad de tener que especificar los tipos en las declaraciones, eso sí, siempre que estemos asignando algún valor a la variable.

```
var producto = "Alfajor Capitán del Espacio";

Console.WriteLine("{0} es de tipo {1}", nameof(producto), producto.GetType().Name);

var capas = 3;

Console.WriteLine("{0} es de tipo {1}", nameof(capas), capas.GetType().Name);

var precio = 99.99M;

Console.WriteLine("{0} es de tipo {1}", nameof(precio), precio.GetType().Name);

var peso = 40F;
```

```
Console.WriteLine("{0} es de tipo {1}", nameof(peso), peso.GetType().Name);

var stock = 1000L;

Console.WriteLine("{0} es de tipo {1}", nameof(stock), stock.GetType().Name);

var glaseado = true;

Console.WriteLine("{0} es de tipo {1}", nameof(glaseado), glaseado.GetType().Name);

var codigoGusto = 'C';

Console.WriteLine("{0} es de tipo {1}", nameof(codigoGusto),
codigoGusto.GetType().Name);
```

La salida del código anterior es:

```
producto es de tipo String
capas es de tipo Int32
precio es de tipo Decimal
peso es de tipo Single
stock es de tipo Int64
glaseado es de tipo Boolean
codigoGusto es de tipo Char
```

El método `GetType` es heredado de la clase `System.Object` para todos los tipos de dato y permite obtener información en tiempo de ejecución sobre el tipo almacenado en la variable.

> **ADVERTENCIA**
>
> Algunos desarroladores consideran que usar `var` afecta negativamente la legibilidad del código. **Evitá usarlo durante la cursada.**

Valores por defecto

Por defecto los tipos de valor contienen del valor `0` si son numéricos, `false` si son de tipo `bool` y `''` si son de tipo `char`. Por otra parte, si no se inicializan, las variables de tipos de referencia contendrán el valor `null` que indica que esa variable no apunta a ninguna dirección de memoria.

El operador `default` recibe como argumento un tipo de dato y nos devuelve el valor por defecto de ese tipo.

```
Console.WriteLine("Valor por defecto de enteros: {0}", default(int));
Console.WriteLine("Valor por defecto de flotantes: {0}", default(double));
Console.WriteLine("Valor por defecto de booleanos: {0}", default(bool));
Console.WriteLine("Valor por defecto de fechas: {0}", default(DateTime));
Console.WriteLine("Valor por defecto de strings: {0}", default(string));
Console.WriteLine("Valor por defecto de chars: {0}", default(char));
```

La salida del código anterior es:

```
Valor por defecto de enteros: 0
Valor por defecto de flotantes: 0
Valor por defecto de booleanos: False
Valor por defecto de fechas: 1/1/0001 00:00:00
Valor por defecto de strings:
Valor por defecto de chars:
```

Tamaño de tipos

El operador `sizeof()` recibe como argumento un tipo de dato y retorna el número de bytes que usa ese tipo en memoria.

Además, algunos tipos cuentan con las propiedades `MinValue` y `MaxValue` que retornan el valor mínimo y el valor máximo que puede almacenar una variable de cierto tipo.

```
Console.WriteLine("int usa {0} bytes y soporta un rango de valores entre {1} y {2}.",
sizeof(int), int.MinValue, int.MaxValue);
Console.WriteLine("double usa {0} bytes y soporta un rango de valores entre {1} y
{2}.", sizeof(double), double.MinValue, double.MaxValue);
Console.WriteLine("decimal usa {0} bytes y soporta un rango de valores entre {1} y
{2}.", sizeof(decimal), decimal.MinValue, decimal.MaxValue);
```

La salida del código anterior es:

```
int usa 4 bytes y soporta un rango de valores entre -2147483648 y 2147483647.
double usa 8 bytes y soporta un rango de valores entre -1,7976931348623157E+308 y
1,7976931348623157E+308.
decimal usa 16 bytes y soporta un rango de valores entre
-79228162514264337593543950335 y 79228162514264337593543950335.
```

Conversiones de tipos de datos

Implicitas

No interviene el programador (no requieren casteo).

No deberían implicar pérdida de datos.

```
// Los float pueden almacenar números más grandes que los int.
// No hay pérdida de datos.

float entero = 15;
```

Explicitas

Interviene el programador (se quiere un casteo).

Podrían implicar pérdida de datos.

```
// Los double pueden almacenar números más grandes que los int.
// Además los enteros no guardan los decimales.
// Puede haber pérdida de datos.

int entero = (int)15.2;
```

On this page

Operadores lógicos

Los **operadores lógicos** trabajan con valores booleanos. El resultado también será `true` o `false`.

Las tablas de verdad para los operadores lógicos son:

Negación

Cuando usamos el **operador de negación** la expresión retorna si **todos** los operandos son `true`.

El símbolo para el operador OR es `&`.

Operando A	Operando B	Resultado
`false`	`false`	`false`
`true`	`false`	`false`
`false`	`true`	`false`
`true`	`true`	`true`

Operador AND - Conjunción lógica

Cuando usamos el **operador AND** la expresión retorna `true` si **todos** los operandos son `true`.

El símbolo para el operador OR es `&`.

Operando A	Operando B	Resultado
`false`	`false`	`false`
`true`	`false`	`false`
`false`	`true`	`false`
`true`	`true`	`true`

Operador OR - Disyunción lógica

Cuando usamos el **operador OR** la expresión retorna `true` si **al menos uno de** los operandos es `true`.

Operando A	Operando B	Resultado
`false`	`false`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`true`	`true`	`true`

El símbolo para el operador OR es `|`.

Operador XOR - Disyunción exclusiva

Cuando usamos el **operador XOR** la expresión retorna `true` si **sólo uno de** los operandos es `true`.

Operando A	Operando B	Resultado
`false`	`false`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`true`	`true`	`false`

El símbolo para el operador XOR es `^`.

Operadores lógicos condicionales

Los **operadores lógicos condicionales** son similares a los operadores lógicos pero son más eficientes ya que no evalúan toda la expresión si ya está determinado el resultado.

Por ejemplo, sabemos que en una operación de conjunción lógica (AND) ambos operandos tienen que ser verdaderos para que la expresión sea verdadera. Si ya el primer operando se resuelve como false, el

operador condicional lógico AND (`&&`) no evaluará el segundo operando. A diferencia del operador lógico AND (`&`) que evaluará ambos operandos siempre.

```
//Si MetodoUno() es True, entonces NO se evalua MetodoDos()

if (MetodoUno() || MetodoDos())
{ }

//Si MetodoUno() es False, entonces NO se evalua MetodoDos()

if (MetodoUno() && MetodoDos())
{ }
```

> **BUENA PRÁCTICA**
>
> Los operadores lógicos condicionales como `&&` y `||` hacen a nuestras aplicaciones más eficientes. **Son los que usaremos durante la cursada**. Sin embargo, debemos ser conscientes de la diferencia con los operadores lógicos para evitar introducir errores sutiles por no conocer como funcionan.

Sentencias de selección

Las **sentencias de selección** permiten a una aplicación ramificarse y tomar distintos flujos de código dependiendo de un criterio o condición.

En C# existen dos sentencias de selección `if` y `switch`, y la segunda es sólo para hacer más legible el código en algunos escenarios.

Sentencia if

La **sentencia if** evalúa una expresión booleana. Si la expresión resulta en `true`, se ejecutará el bloque de código asociado al `if`.

```
// Si x es menor a 10 se ejecuta MetodoUno() de lo contrario no pasa nada.

if (x < 10)
{
    MetodoUno();
}
```

El bloque `else` es opcional y sólo se ejecuta si la expresión del `if` resulta en `false`.

```
// Si x es menor a 10 se ejecuta MetodoUno().
// Si x es mayor o igual a 10 se ejecuta MetodoDos().

if (x < 10)
{
    MetodoUno();
}
else
{
    MetodoDos();
}
```

Los bloques `if` se pueden anidar. También se pueden combinar con otros `if` usando `else if`, siempre y cuando las condiciones sean mutuamente excluyentes.

```
// Si x es menor a 10 se ejecuta MetodoUno().
// Si x está entre 10 y 20 incluido se ejecuta MetodoDos().
// Si ninguno de los casos se cumple se ejecuta MetodoTres().

if (x < 10)
{
    MetodoUno();
}
else if (x >= 20)
{
    MetodoDos();
}
else
{
    MetodoTres();
}
```

Sobre no usar llaves en las sentencias if

Existe la posibilidad de no usar llaves con las sentencias `if` siempre que dentro del bloque haya una sola línea de código.

```
if (edad >= 18)
    Console.WriteLine("Es mayor de edad");
else
    Console.WriteLine("Es menor de edad");
```

ADVERTENCIA

> No usar llaves no hace al código más eficiente ni trae ningún tipo de ventaja, al contrario, hace al código menos mantenible y puede introducir errores.
>
> Por ejemplo, en 2012 se lanzó iOS 6 y trajo un bug que afectaba su sistema de encriptación Secure Sockets Layer (SSL). Los usuarios estuvieron expuestos a vulnerabilidades de seguridad durante 18 meses porque una verificación importante estaba siendo accidentalmente salteada. Todo por no usar llaves en un bloque if.

Sentencia switch

La sentencia `switch` compara una variable específica contra una lista de posibles valores identificados en las sentencias `case`.

Todas las sentencias `case` deben finalizar con la palabra clave `break`.

```
int a = 0;

switch  (a)
{
    case 0:
        MetodoUno();
        break;

    case 1:
        MetodoDos();
        break;
}
```

El `switch` del ejemplo evalua la variable `a` y compara su valor con cada uno de los literales en las sentencias `case`. En este caso se ejecutará el `MetodoUno()`.

Al final del swift se puede agregar una sentencia `default` que se ejecuta si la expresión evaluada no coincide con ninguno de los literales en las sentencias `case`.

```
int a = 9;

switch  (a)
{
    case 0:
        MetodoUno();
        break;

    case 1:
```

```
        MetodoDos();
        break;

    default:
        MetodoTres();
        break;
}
```

En el ejemplo anterior se ejecutará el `MetodoTres()`.

On this page

Sentencias de iteración

Las **sentencias de iteración** permiten repetir la ejecución de un bloque mientras una condición se cumpla. También permiten recorrer los elementos de colecciones y arrays.

En C# existen dos sentencias de selección `if` y `switch`, y la segunda es sólo para hacer más legible el código en algunos escenarios.

Sentencia while

Sentencia do-while

Sentencia for

Sentencia foreach

On this page

Trabajando con la consola

Las aplicaciones de consola se basan en texto y corren sobre la línea de comandos. Se suelen utilizar para ejecutar tareas simples y concretas. Entre sus tareas más habituales se encuentra escribir y leer datos.

> **INFORMACIÓN**
>
> Prácticamente toda la interacción con la consola se logra a través de los métodos y propiedades de la clase `Console`. Para más información sobre esta clase visita la documentación oficial.

Salida de datos por consola

Para mostrar texto se utilizan los métodos `Write` y `WriteLine` de la clase `Console`, cuya única diferencia es que el último genera un salto de línea después de imprimir el texto.

Reciben como argumento el texto a mostrar en la consola.

```
Console.WriteLine("Texto con salto de línea.");
Console.Write("Texto sin ");
Console.Write("salto de línea.");
```

La salida del código anterior es:

```
Texto con salto de línea.
Texto sin salto de línea.
```

Formato compuesto

Ambos métodos soportan una característica llamada **formatos compuestos** que consisten es una lista de valores y una cadena de formato compuesto. Una **cadena de formato compuesto** está formada por texto fijo combinado con **marcadores de posición** que corresponden con los elementos de la lista.

El resultado de la operación de formato es una cadena de texto compuesta por el texto fijo combinado con la representación en texto de los objetos de la lista.

```
string facultad = "UTN Fra";
```

```
string materia = "Programación II";
short anio = 2021;
byte cuatrimestre = 2;

Console.WriteLine("Estamos en la materia {0} de {1}. Es el año {2} y el cuatrimestre {3}.", materia, facultad, anio, cuatrimestre);
```

La salida del código anterior es:

```
Estamos en la materia Programación II de UTN Fra. Es el año 2021 y el cuatrimestre 2.
```

Los marcadores de posición (`{0}`, `{1}`, `{2}`, `{3}`) corresponden a la posición de las variables desde el segundo argumento del método en adelante. Es decir, `{0}` corresponde a la representación en texto de `facultad`, `{1}` corresponde a la representación en texto de `materia`, y así sucesivamente. Podemos observar que las posiciones se empiezan a contar partiendo del número cero.

Si cambiaramos de orden las variables o los marcadores, daría un resultado completamente distinto.

```
Console.WriteLine("Estamos en la materia {1} de {0}. Es el año {3} y el cuatrimestre {2}.", materia, facultad, anio, cuatrimestre);
```

La salida del código anterior es:

```
Estamos en la materia UTN Fra de Programación II. Es el año 2 y el cuatrimestre 2021.
```

En el ejemplo anterior se cambia el número de los marcadores, y como representa a las posición de las variables podemos observar que nos muestra completamente distinto. Lo mismo sucede en el siguiente ejemplo donde lo que cambia es el orden de la lista de variables.

```
Console.WriteLine("Estamos en la materia {0} de {1}. Es el año {2} y el cuatrimestre {3}.", cuatrimestre, anio, facultad, materia);
```

La salida del código anterior es:

```
Estamos en la materia 2 de 2021. Es el año UTN Fra y el cuatrimestre Programación II.
```

Aplicar formatos a cadenas de texto

A los formatos compuestos de los métodos de entrada y a cualquier `string` le podemos aplicar formatos especiales o personalizados.

Por ejemplo, si queremos dibujar una tabla con el nombre de distintos productos alineados a la izquierda y dentro de una columna de 10 caracteres, y los precios de cada uno alineados a la derecha con separador de miles, con dos decimales y con una columna de 6 caracteres:

```
string placaDeVideo = "Nvidia RTX 3080";
decimal precioPlacaVideo = 344663.36M;
string procesador = "Ryzen 7 5800x";
decimal precioProcesador = 63595M;

Console.WriteLine("{0,-20} {1,10}", "Producto", "Precio");
Console.WriteLine("{0,-20} {1,10:N2}", placaDeVideo, precioPlacaVideo);
Console.WriteLine("{0,-20} {1,10:N2}", procesador, precioProcesador);
```

La salida del código anterior es:

```
Producto                 Precio
Nvidia RTX 3080      344.663,36
Ryzen 7 5800x         63.595,00
```

Primero se debe escribir entre llaves la posición o índice de la variable que queremos imprimir. `{1}` indica que allí se deberá insertar la representación en texto de la variable que está en la segunda posición.

Se puede definir el ancho del campo agregando una coma y un valor. Si el valor es positivo la cadena se alinea a la derecha, si es negativo se alinea a la izquierda. `{1,10}` aplica al valor de la variable en la segunda posición un ancho de columna de 10 caracteres y alineación a la derecha.

También se puede aplicar un formato específico agregando al marcador el símbolo de dos puntos `:` seguido de la cadena de formato. `{1,10:N2}` aplica al valor del marcador el formato `N2` que representa un número con separadores de dígitos y hasta 2 decimales.

Como formula general tenemos:

```
{N[,M][:Formato]}
```

- `N` será el número que representa a la posición o índice del parámetro, empezando por cero.
- `M` será el ancho usado para mostrar el parámetro, el cual se rellenará con espacios. Si `M` es negativo se justificará a la izquierda, si es positivo a la derecha. Es opcional.

- `Formato` será una cadena que indicará un formato a aplicar sobre ese parámetro. Es opcional.

> **INFORMACIÓN**
>
> Para conocer más detalles sobre los tipos de formato aplicables visita la documentación oficial.

Aplicar un formato numérico estandar

¿Qué pasa si quiero mostrar un precio y quiero que los números siempre se impriman con 2 decimales y con el signo de la moneda? Cambio la última línea y aplico un **formato numérico estandar**.

```
Console.WriteLine("Se ingreso el precio {0:C2}, con IVA incluido suma ${1:C2}",
precioSinIva, precioFinal);
```

`:C2` es el formato que se aplica sobre el valor en el marcador `0`. `C` se utiliza para formato de monedas y el `2` indica la cantidad de digitos desde el punto decimal que quiero mostrar.

La salida del código anterior es:

```
Ingrese un precio: 12
Se ingreso el precio $ 12,00, con IVA incluido suma $ 14,52
```

> **INFORMACIÓN**
>
> Para conocer todos los formatos numéricos estandard visita la documentación oficial.

Aplicar un formato numérico personalizado

Otra forma de hacer lo mismo es con un **formato numérico personalizado**

```
Console.WriteLine("Se ingreso el precio {0:$#.00}, con IVA incluido suma ${1:$#.00}",
precioSinIva, precioFinal);
```

El signo `$` es un literal fijo. `#` representa un digito siempre que exista, de lo contrario muestra un string vacío. `.` es el punto decimal. `00` indica hasta dos decimales, y si hay menos o no existen rellena con ceros.

La salida del código anterior es:

```
Ingrese un precio: 12
```

```
Se ingreso el precio $12,00, con IVA incluido suma $14,52
```

> **INFORMACIÓN**
>
> Para conocer todos los formatos numéricos personalizados visita la documentación oficial.

Secuencias de escape

Dentro de los `string` se puede incluir **secuencias de escape** que se utilizan para representar caracteres especiales. Los caracteres de escape vienen acompañados con el prefijo `\` (contra barra).

Secuencia de escape	Descripción
`\'`	Comilla simple
`\"`	Comilla doble
`\\`	Contra barra
`\n`	Nueva línea / Salto de línea
`\t`	Tabulación horizontal

```
Console.WriteLine("Texto con salto de línea.\n")
Console.WriteLine("\tTexto tabulado horizontalmente.");
Console.WriteLine("Pongo \"comillas dobles\" a mi texto y también \'comillas
simples\'.");
Console.WriteLine("Para trabajar con caracteres especiales hay que usar \\.");
```

La salida del código anterior es:

```
Texto con salto de línea.

        Texto tabulado horizontalmente.
Pongo "comillas dobles" a mi texto y también 'comillas simples'.
Para trabajar con caracteres especiales hay que usar \.
```

Existe una forma de escapar todos los caracteres especiales de un texto con el prefijo `@`, que también sirve para trabajar con texto multi-línea.

```
Console.WriteLine(@"La ruta al archivo es: C:\\usuario\documentos\texto.txt");
Console.WriteLine(@"
Un texto
en más de una línea
es posible con C#
anteponiendo @ al literal de string.
");
```

La salida del código anterior es:

```
La ruta al archivo es: C:\\usuario\documentos\texto.txt

Un texto
en más de una línea
es posible con C#
anteponiendo @ al literal de string.
```

Entrada de datos por consola

Leer una línea de texto

Para tomar un dato ingresado por el usuario se usa el método `ReadLine` de la clase `Console`.

```
Console.Write("Ingrese una fruta: ");
string fruta = Console.ReadLine();

Console.WriteLine("Se ingreso la fruta {0}", fruta);
```

La salida del código anterior es:

```
Ingrese una fruta: Banana
Se ingreso la fruta Banana
```

Leer números

El método `ReadLine` devuelve siempre un `string`. Si trabajamos con números tendremos que convertirlos a un tipo numérico.

```
Console.Write("Ingrese un precio: ");
```

```
string precioIngresado = Console.ReadLine();

decimal precioSinIva = decimal.Parse(precioIngresado);

decimal precioFinal = precioSinIva + precioSinIva * 0.21M;

Console.WriteLine("Se ingreso el precio ${0}, con IVA incluido suma ${1}",
precioSinIva, precioFinal);
```

La salida del código anterior es:

```
Ingrese un precio: 12
Se ingreso el precio 12, con IVA incluido suma 14,52
```

Leer una tecla

Para obtener la tecla presionada por el usuario se utiliza `ReadKey` de la clase `Console`. Este método espera a que el usuario presione una tecla o combinación de teclas y retorna un objeto de tipo `ConsoleKeyInfo`.

A partir de la variable donde almacenamos el objeto de tipo `ConsoleKeyInfo` podremos acceder a las siguientes propiedades:

- `Key`: retorna un objeto de tipo `ConsoleKey` que representa a la tecla presionada.
- `KeyChar`: retorna como tipo `char` el caracter relacionado a la tecla presionada (siempre que exista, de lo contrario retorna espacio vacío).
- `Modifiers`: retorna un objeto de tipo `ConsoleModifiers` que representa a los modificadores que se hayan aplicado en la combinación de teclas. Ejemplos de modificadores son `SHIFT`, `CTRL`, `ALT`, etc.

Si presionamos, por ejemplo, la tecla `M` se imprimirá el caracter en la consola. Si al método `ReadKey` le pasamos como argumento `true` interceptará el caracter presionado y no lo mostará.

```
Console.WriteLine("Presione una tecla o combinación de teclas:");

ConsoleKeyInfo teclaPresionada = Console.ReadKey(true);

ConsoleKey tecla = teclaPresionada.Key;
char caracter = teclaPresionada.KeyChar;
ConsoleModifiers modificadores = teclaPresionada.Modifiers;

Console.WriteLine("Tecla: {0}. Caracter: {1}. Modificadores: {2}", tecla, caracter,
```

```
modificadores);
```

Al presionar la tecla `M`, la salida del código anterior es:

```
Tecla: M. Caracter: m. Modificadores: 0
```

Si presionamos simultáneamente la tecla `Shift` y la tecla `M`, la salida será:

```
Tecla: M. Caracter: M. Modificadores: Shift
```

Modificando la consola

La clase `Console` tiene propiedades que nos permiten modificar cómo se mostrará la salida de texto.

Limpiar la consola

Para borrar el contenido de la consola se puede utilizar el método `Clear` de la clase `Console`.

```
Console.Clear();
```

Cambiar el color del texto

Para cambiar el color del texto, se tiene que asignar un nuevo valor a la propiedad `ForegroundColor` de la clase `Console`. Cada color es representado por un número. Para ayudarnos existe el enumerado `ConsoleColor` que contiene estos números representados por nombres descriptivos.

Por ejemplo, para cambiar el color del texto a rojo se deberá escribir:

```
Console.ForegroundColor = ConsoleColor.Red;
```

Cambiar el color de fondo del texto

Para cambiar el color de fondo del texto, se tiene que asignar un nuevo valor a la propiedad `BackgroundColor` de la clase `Console`. También usaremos el enumerado `ConsoleColor` para obtener el código del color.

Por ejemplo, para cambiar el color de fondo del texto a azul se deberá escribir:

```
Console.BackgroundColor = ConsoleColor.Blue;
```

Cambiar el título de la consola

Para cambiar el título que se muestra en la ventana de la consola se debe asignar el nuevo texto a la propiedad `Title` de la clase `Console`.

```
Console.Title = "Mi primer programa en C#";
```

Cerrar la consola

Para cerrar la consola se utiliza el método `Exit` de la clase `Environment`.

```
Environment.Exit(0);
```

`Exit` recibe como argumento un código de salida. El código `0` se utiliza para indicar que el proceso finalizó correctamente, otros números se pueden usar para señalizar errores de ejecución.

Emitir un sonido

El método `Beep` de la clase `Console` nos permite emitir un sonido a través del altavoz de la consola. Opcionalmente, se le puede pasar como argumentos la frecuencia del sonido en hertz y la duración en milisegundos.

```
Console.Beep(); // Sonido por defecto.

Console.Beep(1000, 1500); // Sonido de 1000hz durante 1 segundo y medio.
```

> ⚠ **ADVERTENCIA**
>
> El método `Beep` no funciona en todas las plataformas.

Cuestionario - Introducción a .NET y C#

.NET

1. ¿Qué es .NET? ¿Cuáles son sus principales características?
2. ¿Cómo se categorizan las versiones de .NET? Explique su relación con las políticas de soporte y mantenimiento de dichas versiones.
3. ¿Qué componentes forman parte de una implementación de .NET? Describa brevemente cada uno.
4. ¿Qué es el Common Language Runtime?
5. ¿Qué es la Base Class Library?
6. ¿En qué se diferencia un marco de trabajo (framework) de una biblioteca (library)?
7. Describa los estados y etapas del proceso de compilación de .NET.
8. Explique y compare tiempo de compilación y tiempo de ejecución.

C#

1. Enumere y describa las características de C#.
2. Explique la diferencia entre lenguajes de tipado estático y lenguaje de tipado dinámico.
3. Defina sintaxis, semántica y vocabulario de una lenguaje de programación.
4. Defina los siguientes términos: Sentencia (statement), variable y expresión (expression). ¿Cómo se relacionan?
5. ¿En qué se diferencian variables y constantes?
6. ¿Qué es el punto de entrada (entry point)? ¿Cuál es en los programas de C#?
7. ¿Qué es el Common Type System (CTS)?
8. Explique las diferencias entre los tipos de referencia y los tipos de valor.
9. ¿Cuáles son las categorías de tipos de datos de .NET? Clasifique en tipos de valor y tipos de referencia.
10. ¿Qué es un alias?
11. Ordene los siguientes tipos de datos del menos preciso al más preciso: decimal, float, double.
12. ¿Qué son los tipos object y dynamic? ¿En qué se diferencian?
13. ¿Para qué se usan los operadores `nameof()` y `sizeof()`?
14. ¿Cuál es el valor por defecto de los tipos de datos en .NET?
15. ¿En qué consiste el cortocircuito lógico de los operadores `&&` y `||`?

16. ¿Qué es el formato compuesto? ¿Cómo se aplica en el método WriteLine? Mencione cómo configurar el ancho de los campos, justificación a la izquierda o derecha, formatos estandar y formatos personalizados.

*Last updated on **8/23/2021** by **mauriciocerizza***

Ejercicio I01 - Máximo, mínimo y promedio

Consigna

Ingresar 5 números por consola, guardándolos en una variable escalar. Luego calcular y mostrar: el valor máximo, el valor mínimo y el promedio.

> **INFORMACIÓN**
>
> Un **escalar** es una constante o variable que contiene un dato atómico y unidimensional. En contraposición al concepto de escalar, están los conceptos de array, lista y objeto, que pueden tener almacenado en su estructura más de un valor. Ver en Wikipedia

Resolución

	Video		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I02 - Error al cubo

Consigna

Ingresar un número y mostrar el cuadrado y el cubo del mismo. Se debe validar que el número sea mayor que cero, caso contrario, mostrar el mensaje: "*ERROR. ¡Reingresar número!*".

> **IMPORTANTE**
>
> Utilizar el método `Pow` de la clase `Math` para calcular las potencias.

Resolución

	**Video**		**Código**

*Last updated on **8/23/2021** by **mauriciocerizza***

Ejercicio I03 - Los primos

Consigna

Mostrar por pantalla todos los números primos que haya hasta el número que ingrese el usuario por consola.

Validar que el dato ingresado por el usuario sea un número.

Volver a pedir el dato hasta que sea válido o el usuario ingrese "*salir*".

Si ingresa "*salir*", cerrar la consola.

Al finalizar, preguntar al usuario si desea volver a operar. Si la respuesta es afirmativa, iterar. De lo contrario, cerrar la consola.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración, selectivas y el operador módulo (%).

Resolución

	Video		**Código**

*Last updated on **8/23/2021** by **mauriciocerizza***

Ejercicio I04 - Un número perfecto

Consigna

Un número perfecto es un entero positivo, que es igual a la suma de todos los enteros positivos (excluido el mismo) que son divisores del número.

El primer número perfecto es 6, ya que los divisores de 6 son 1, 2 y 3; y 1 + 2 + 3 = 6.

Escribir una aplicación que encuentre los 4 primeros números perfectos.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración y selectivas.

> **INFORMACIÓN**
>
> Para conocer más sobre los números perfectos, consultar el siguiente enlace a Wikipedia.

Resolución

	Video		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I05 - Tirame un centro

Consigna

Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales.

El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1; 2; 3; 4; 5) y (7; 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595.

Se pide elaborar una aplicación que calcule los centros numéricos entre 1 y el número que el usuario ingrese por consola.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración y selectivas.

Resolución

	**Video**		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I06 - Años bisiestos

Consigna

Escribir un programa que determine si un año es bisiesto.

Un año es bisiesto si es múltiplo de 4. Los años múltiplos de 100 no son bisiestos, salvo si ellos también son múltiplos de 400. Por ejemplo: el año 2000 es bisiesto pero 1900 no.

Pedirle al usuario un año de inicio y otro de fin y mostrar todos los años bisiestos en ese rango.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración, selectivas y el operador módulo (%).

Resolución

	**Video**		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I07 - Recibo de sueldo

Consigna

Se debe pedir el ingreso por teclado del *valor hora*, el *nombre*, la *antigüedad* (en años) y la *cantidad de horas* trabajadas en el mes de ***N*** cantidad de empleados de una fábrica.

Se pide calcular el importe a cobrar teniendo en cuenta que el total (que resulta de multiplicar el valor hora por la cantidad de horas trabajadas), hay que sumarle la cantidad de años trabajados multiplicados por $150, y al total de todas esas operaciones restarle el 13% en concepto de descuentos.

Mostrar el recibo correspondiente con el *nombre*, la *antigüedad*, el *valor hora*, el *total a cobrar en bruto* y el *total a cobrar neto* de todos los empleados ingresados.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración y selectivas.
>
> No es necesario ni se deben utilizar vectores/arrays.

Resolución

	Video		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I08 - Dibujando un triángulo rectángulo

Consigna

Escribir un programa que imprima por consola un triángulo como el siguiente:

```
*
***
*****
*******
*********
```

El usuario indicará cuál será la altura del triángulo ingresando un número entero positivo. Para el ejemplo anterior, la altura ingresada fue de 5.

> **IMPORTANTE**
>
> Utilizar de iteración y selectivas.

Resolución

	**Video**		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***

Ejercicio I09 - Dibujando un triángulo equilátero

Consigna

Escribir un programa que imprima por consola un triángulo como el siguiente:

```
    *
   ***
  *****
 *******
*********
```

El usuario indicará cuál será la altura del triángulo ingresando un número entero positivo. Para el ejemplo anterior, la altura ingresada fue de 5.

> **IMPORTANTE**
>
> Utilizar sentencias de iteración y selectivas.

Resolución

	**Video**		**Código**

*Last updated on **8/21/2021** by **mauriciocerizza***