

APRENDE A PROGRAMAR con C#



Héctor de León

Aprender a programar con C#

Un libro lleno de conceptos importantes para programadores

Héctor de León

Aprender a programar con C#

Un libro lleno de conceptos importantes para programadores

Héctor de León

ISBN Versión Impresa: 978-607-29-5203-4

ISBN Versión Digital: 978-607-29-5204-1

Reservados todos los derechos. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de los titulares del copyright. La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.

© 2023 Héctor de León

Índice

Contenido

[Aprender a programar con C#](#)

[Índice](#)

[Agradecimientos](#)

[Sobre Héctor de León](#)

[Prólogo](#)

[1. Introducción](#)

[¿Por qué C#?](#)

[El nacimiento del lenguaje de programación C#](#)

[Diferencia entre C# y .NET](#)

[¿Cómo programar en C#?](#)

[2. Lo básico de programación](#)

[¿Qué es programar?](#)

[Variables](#)

[Tipos de Dato](#)

[Estructuras de control](#)

[Sentencia if y else](#)

[Operadores de comparación](#)

[Operadores lógicos booleanos](#)

[Sentencia Switch](#)

[Bucle While](#)

[Bucle Do While](#)

[Bucle For](#)

[Funciones](#)

[3. Array](#)

[¿Qué es un array?](#)

[Recorrido de un array](#)

[Funciones que regresan un array](#)

[Alternativa a Switch, utilizando Array](#)

[4. Programación Orientada a Objetos Básica](#)

[Paradigmas en la programación](#)

[¿Qué es la Programación Orientada a Objetos?](#)

[Clases y Objetos](#)

[Constructor](#)

[Herencia](#)

[Constructores en Herencia](#)

[Encapsulamiento](#)

[Encapsulamiento en Herencia](#)

[Campos y Propiedades](#)

[Static](#)

[Sobrecarga](#)

[Sobreescritura](#)

[Ventajas de la programación orientada a objetos](#)

[5. Conceptos importantes en la Programación](#)

[Estructuras de datos](#)

[Ámbito](#)

[Null](#)

[Parámetros por referencia](#)

[Parámetros por valor](#)

[Inmutabilidad](#)

[Azúcar sintáctica](#)

[6. Programación Funcional](#)

[¿Qué es la programación funcional?](#)

[Programación imperativa vs programación declarativa](#)

[Función de primera clase](#)

[Función orden superior](#)

[Función pura](#)

[Funciones anónimas \(Expresiones lambda\)](#)

[Tipado de funciones](#)

[Recursividad](#)

[Closures \(Clausuras\)](#)

[Curricación](#)

[Ventajas de la programación funcional](#)

[7. Programación Orientada a Objetos Avanzada](#)

[Polimorfismo](#)

[Polimorfismo por herencia](#)

[Interfaces](#)

[Generics](#)

[Foreach](#)

[Abstracción](#)

[Clase Abstracta](#)

[Clase Abstracta vs Interface](#)

[8. Conceptos extras, pero importantes](#)

[Excepciones](#)

[Serialización](#)

[JSON](#)

[9. Consejos finales](#)

Agradecimientos

Agradezco a mis perros Forky, Camila, Charlotte y mis gatos Pikachu, Candy, Tomy, Concha y Concha 2 que siempre han sido parte de mi motivación por realizar proyectos como este, a mis padres que me dieron la oportunidad de adentrarme en el mundo de la Programación, a pesar de que hasta la fecha no saben a qué me dedico, pero se animaron a pagarme un curso hace más de 20 años y aquí sigo, pero sobre todo agradezco a Itza que siempre ha estado a mi lado en cada proyecto en el cual he incursionado.



Sobre Héctor de León

Escribir sobre mi es raro, pero tratare de ser breve, soy Héctor de León, programador de años, ya muchos, y desde muy pequeño me adentré en este mundo de escribir código gracias a que mis padres me pagaron un curso de programación.

Desde muy temprana edad comencé a trabajar como programador, ya que en aquel entonces había muy pocos, y ya estando trabajando me anime a estudiar en la Universidad, por lo cual soy Ingeniero en Computación.

He estado trabajando como programador sin descanso desde entonces, ya más de 20 años, tanto en empresas y posterior desde hace algunos años en mis propios proyectos.

Tengo un canal de YouTube relacionado a la Programación desde el 2018 (aunque la fecha dice 2016 nunca subí nada en ese año), que cuenta con más de 100,000 suscriptores, en dicho canal creo contenido gratuito, el canal se llama hdeleon.net.

He creado algunos cursos de paga, los cuales van desde estructura de datos, patrones de diseño, y en mi experiencia compartiendo contenido, me hacía falta escribir un libro.

Y bueno aquí estamos, escribiendo este libro.

Prólogo

Este para mi es un nuevo logro, el escribir un libro en mi vida ha sido uno de mis objetivos desde hace muchos años, y hoy por fin veo terminado este trabajo.

He querido plasmar mis más de 20 años como programador en este libro, explicando conceptos que siento yo que son obligados conocer por todo programador.

El objetivo que tiene este libro es tomar el lenguaje de programación C# como referencia para ver conceptos importantes de programación.

En sí, el fin es que aprendas a utilizar el lenguaje C# y a su vez otros conceptos importantes que aplica a cualquier otro lenguaje de programación.

Me he centrado en dar un temario que va desde tópicos básicos a temas avanzados, con los cuales podré explicarte teoría y lo más importante: donde esta teoría aplica y es valiosa.

Así que espero te sirva el contenido de este libro en tu vida diaria como programador.

¡Buen viaje!

Héctor de León - 2023

1. Introducción

¿Por qué C#?

El lenguaje de programación C# es un lenguaje bastante versátil, dejando de lado que es multiparadigma (orientado a objetos, funcional, genérico, reflexivo, estructurado, dirigido por eventos), el lenguaje por sí mismo sirve para crear aplicaciones de todo tipo imaginado que va desde: sitios web, aplicaciones de escritorio, apps, servicios web, videojuegos y muchas otras cosas más, aparte de que una vez aprendido la curva de aprendizaje de las distintas tecnologías que comparten este lenguaje es demasiado pequeña, es decir, una vez que aprendes C# con alguna tecnología por ejemplo desarrollo web, aprender a crear videojuegos o crear aplicaciones móviles con este mismo lenguaje te resultará bastante fácil.

He seleccionado este lenguaje por dos cosas, una es que es un lenguaje de programación el cual dominó, he trabajado años con este lenguaje creando todo tipo de proyectos, y la segunda cosa es que C# Es un lenguaje de programación flexible, con el cual podemos hacer cosas de manera estricta y también de manera no estricta. Por lo cual, nos permite adentrarnos en varios conceptos los cuales podrás replicar tanto en lenguajes de programación como JavaScript o Python los cuales son muy flexibles y también otros conceptos más estrictos que podemos relacionar a lenguajes como Java o TypeScript.

Así tomaremos C# como lenguaje de programación para explicar conceptos, y a su vez, aprenderás a cómo escribir dichos conceptos con este lenguaje.

El nacimiento del lenguaje de programación C#

Hablar de historia siempre suena tedioso, pero creo que siempre es importante saber por lo menos los orígenes de las herramientas que estamos utilizando.

El lenguaje de programación C# nace en el año 2000, el creador de este lenguaje de programación es Anders Hejlsberg (involucrado también en la creación de TypeScript y Turbo Delphi).

Antes de que C# se llamará C#, su nombre era SMC (Simple Managed C), pero este nombre debió cambiarse por problemas de marca, y pasó a llamarse C#.

El nombre C# fue tomado de la nota musical **Do Sostenido**, Do sostenido es descrito como la letra C, cuando le es agregado el símbolo # a la letra C, Do se convierte en Do Sostenido, medio tono más elevado. C# tomando esta metáfora, lo hace para indicar que C# es superior al lenguaje de programación C++, un lenguaje que también había hecho una analogía parecida con el lenguaje C asignando dos símbolos ++ después de la letra C.

C# fue ideado como un lenguaje de programación orientado a objetos (más adelante explicaremos este concepto), muy parecido a Java en un inicio (hoy no tanto), C# nació como una competencia a Java ya que este lenguaje predomina al final de los años 90s.

El lenguaje de programación C# ha tenido avances con los años tomando características nuevas, C# nació como un lenguaje muy estricto, pero con el tiempo tomó características que lo hicieron flexible, pero sin perder su esencia.

Diferencia entre C# y .NET

Hay cierta confusión cuando las personas desean adentrarse a programar en C#, pasa muchas veces que se confunde C# con .NET, siendo cosas distintas.

C# es un lenguaje de programación que sirve para codificar, cuenta con una sintaxis definida.

En cambio, .NET es un marco de trabajo, mejor conocidos como Frameworks, el cual cuenta con un entorno para crear desarrollos, ejecutarlos y administrarlos, también cuenta con un conjunto de funcionalidades ya hechas y puestas en bibliotecas.

Puedes utilizar el lenguaje de programación C# para crear proyectos utilizando las funcionalidades ya establecidas en .NET, pero no sólo puedes hacerlo con C#, también podrías optar por Visual Basic o F#, lenguajes de programación que pueden funcionar con .NET.

C# es el lenguaje de programación más utilizado en el marco de trabajo de .NET.

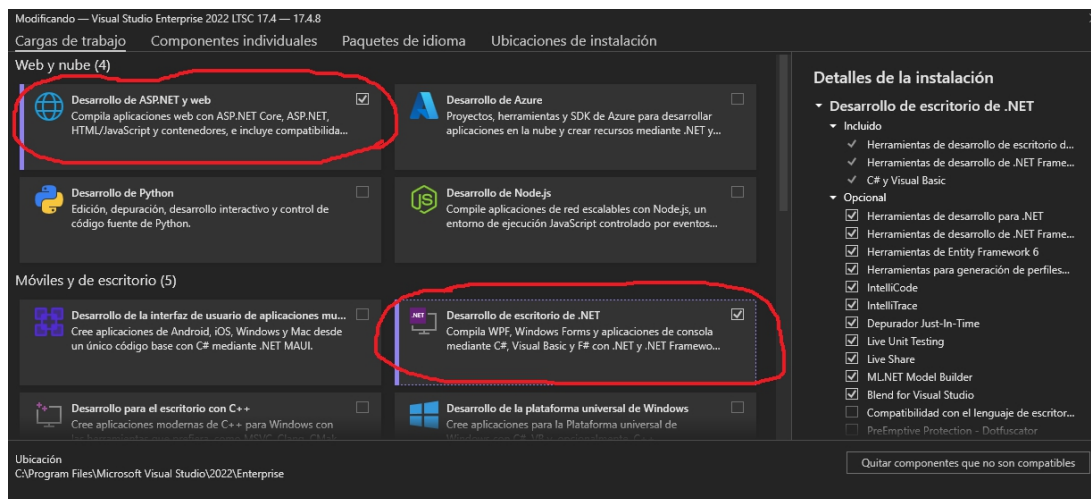
¿Cómo programar en C#?

Para programar en C# puedes lograrlo por muchos medios, pero en este libro nos centraremos en el más fácil, instalar Visual Studio 2022 (No confundir con Visual Studio Code).

Para instalar Visual Studio 2022 (O si hay uno más reciente instalarlo), basta con ir a <https://visualstudio.microsoft.com/> y dar clic en "Descargar" y seleccionar Community el cual es GRATIS:



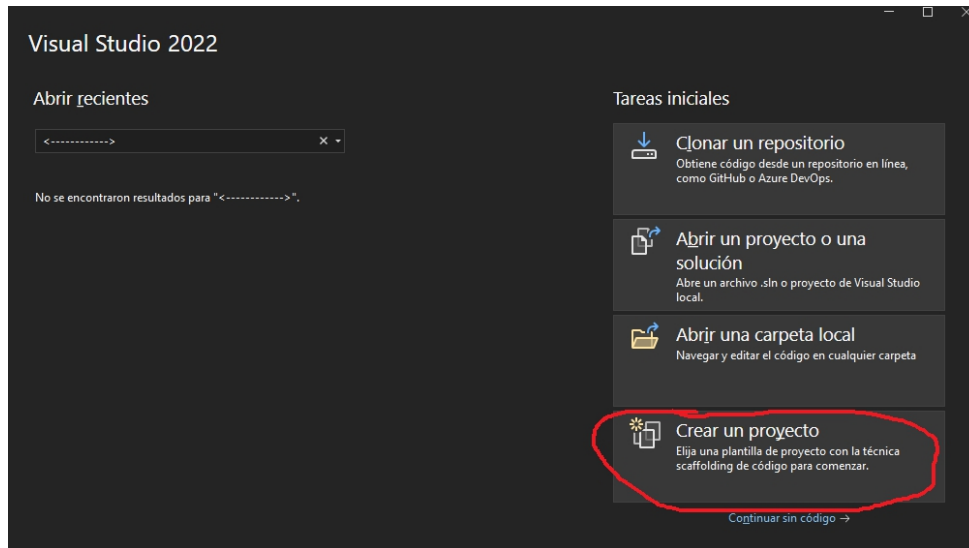
Una vez dado el clic se descargará un archivo que se instala como cualquier otro programa, lo importante es que al llegar a esta sección selecciones las opciones: “Desarrollo de ASP y web” y “Desarrollo de escritorio de .NET”.



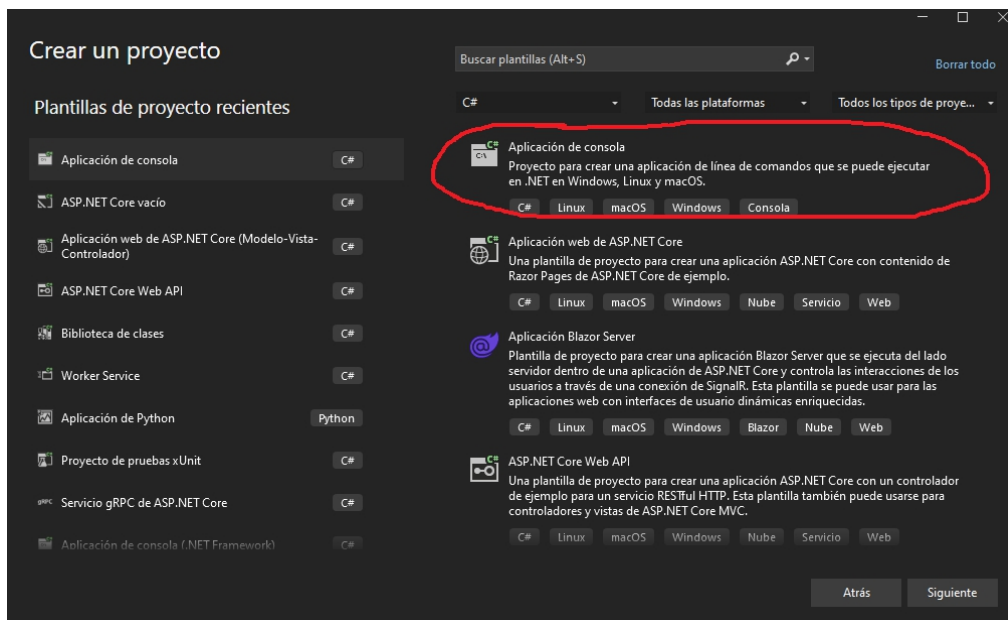
Esas dos opciones estando seleccionadas te permitirán crear proyectos web y proyectos de escritorio.

En este libro trabajaremos con proyectos de tipo consola el cual aparecerá al tener esas opciones instaladas, pero manejaremos funcionalidades que vienen incluidas en las secciones antes seleccionadas.

Una vez instalado, para crear un proyecto basta con que abras Visual Studio 2022 y le des clic en “Crear un proyecto”:



Posterior seleccionar el tipo de proyecto “Aplicación de consola” y dar clic en el botón “Siguiente”:



Lo siguiente será poner un nombre del proyecto, el que gustes y dar clic en siguiente:

Configure su nuevo proyecto

Aplicación de consola C# Linux macOS Windows Consola

Nombre del proyecto
Curso

Ubicación
C:\Users\strat\source\repos\Cursos 2023

Nombre de la solución ⓘ
Curso

☐ Colocar la solución y el proyecto en el mismo directorio

Atrás Siguiente

Por último, debes seleccionar la versión de .NET, por defecto viene la más nueva, te recomiendo seleccionar la más nueva para que tenga todas las características que veremos en este curso, en mi imagen aparece .NET 7 pero es probable que para cuando leas este libro exista ya una versión más nueva, no importa, selecciona la más nueva:

Información adicional


Aplicación de consola C# Linux macOS Windows Consola

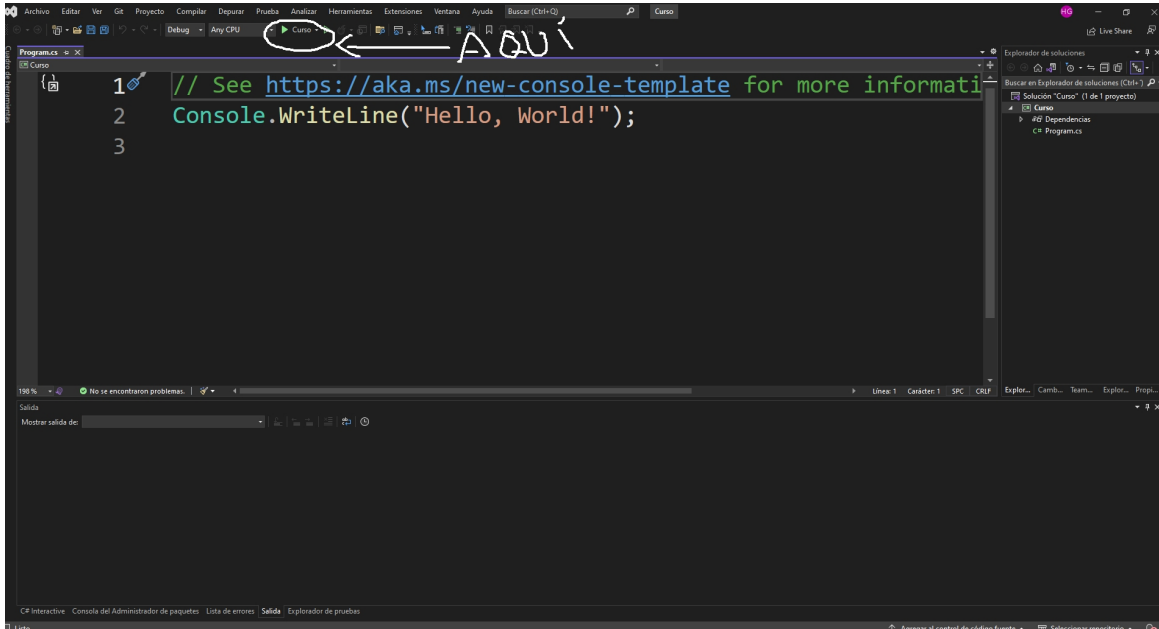
Framework ⓘ
.NET 7.0 (Soporte técnico de términos estándar)

☐ No usar instrucciones de nivel superior ⓘ

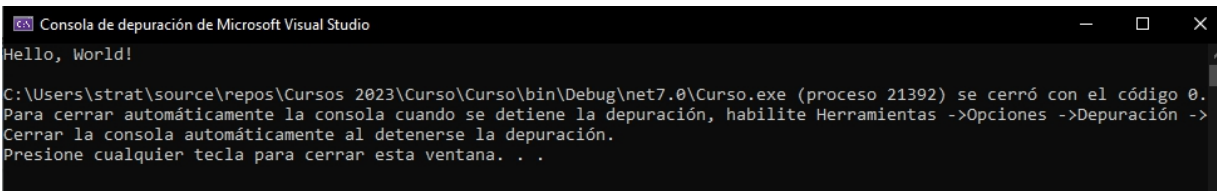
Atrás Crear

Y es todo, el proyecto se creará y estarás listo para escribir código en C# utilizando el marco de trabajo de .NET.

Si das clic en el botón  lo que hará Visual Studio 2022 será compilar tu código y ejecutarlo, da clic para que veas el resultado:



El resultado será una ventana de consola con un mensaje que ha sido escrito en C#:



Y es todo, así es como estaremos probando nuestro código a lo largo de este libro.

2. Lo básico de programación

En esta sección veremos los conceptos básicos que debes conocer para comenzar a programar.

¿Qué es programar?

Programar es enviar un conjunto de instrucciones a una computadora para que esta realice un conjunto de tareas, estas instrucciones se indican gracias a un lenguaje de programación.

Un lenguaje de programación es el medio para encapsular instrucciones singulares que en conjunto realizan una tarea más general. Siendo la tarea general un programa de computadora.

Existen muchos lenguajes de programación, entre ellos Java, JavaScript, Python, TypeScript, y miles más, pero la mayoría de los lenguajes de programación funcionan bajo los mismos conceptos.

Un programador es la persona capacitada en conocer los conceptos y sabe cómo escribirlos en un lenguaje de programación, con dichos conceptos podrá resolver problemáticas con la ayuda de una computadora.

Al final, un programador podrá saber qué conceptos utilizar dependiendo la situación a resolver, en este libro nos enfocaremos a aprender conceptos que te ayudarán sin importar el lenguaje de programación que utilices. Aunque nosotros para explicarlo utilizaremos C#.

Variables

Las variables son una parte fundamental en la programación, imagina que una variable es una caja en la cual puedes almacenar algo, eso que almacenas podrías utilizarlo más adelante para realizar

alguna operación, las variables cumplen ese objetivo, el poder guardar un valor.

Una definición formal de variable es: un contenedor de información que puedes utilizar para almacenar un valor en memoria.

En C# podemos crear variables para almacenar información, esta información serán valores, como números o cadenas de texto, en sí, las variables sirven para guardar información que será utilizada en nuestros programas.

En este libro me encargaré de explicarte el concepto teórico y práctico, y para la parte práctica necesitarás crear un proyecto del tipo consola en Visual Studio.

Una vez tengas tu proyecto de tipo consola procederemos a realizar lo siguiente:

```
// creamos una variable que solo puede guardar números  
int number = 88;  
  
// Línea necesaria para imprimir en pantalla el resultado de una  
// variable  
Console.WriteLine(number);
```

Y procederemos a ejecutar nuestro proyecto para que obtengamos el siguiente resultado:

88

¿Qué es lo que ha pasado? Lo que pasó es que creamos una variable la cual solo puede almacenar números enteros, siendo un número entero, un número que no puede tener decimales, es decir: 1, 4, 56, 10; pero no un número 13.4 o 1.2321.

Una variable en C# debe contener el tipo con el cual vamos a trabajar, en este caso el tipo es un número entero, y la palabra reservada para establecer este tipo es **int**, seguido del nombre que deseamos para nuestra variable, en este ejemplo la llamamos

number, cuando utilizamos la palabra que hemos escrito **number**, hacemos referencia a ella, como lo hicimos en `Console.WriteLine(number)`. Más adelante comprenderás que significa `Console.WriteLine`, así que no te preocupes por ahora con ello, solo vélo como un funcionamiento que nos ayuda a imprimir resultados en pantalla.

También es importante notar que en C# es obligatorio poner punto y coma (;) al final de cada línea de código.

El texto que está posterior a las dos diagonales (`// algo aquí`) es un comentario, este comentario será ignorado al momento de ejecutar nuestro código, pero son útiles para cuando programamos pongamos recordatorios de que es lo que hace nuestro código.

Como podemos ver una variable puede guardar información, pero debemos establecer el tipo de información que va a guardar, por ejemplo, si deseamos guardar una cadena de texto tendríamos que hacer lo siguiente:

```
string text = "Hola soy Héctor de León";  
Console.WriteLine(text);
```

Y tendríamos como resultado:

```
Hola soy Héctor de León
```

Si lo notaste, en este segundo ejemplo, para guardar una cadena de texto en una variable hemos escrito la palabra **string** seguida del nombre de nuestra variable la cual llamamos **name**, pero el valor que hemos puesto lo pusimos entre comillas, es decir "Hola soy Héctor de León".

Cuando trabajamos con cadenas de texto, en la mayoría de lenguajes de programación, va ser común poner el valor entre comillas.

El nombramiento de las variables debe seguir ciertas reglas (dependiendo el lenguaje de programación), en C# la regla es: que siempre debe iniciar por una letra o un guion bajo (_), posteriormente puede tener más letras, guiones bajos (_) y números, y no puedes dejar espacio en blanco en su nombre.

Como consejo en nombramiento de variables es que el nombre te indique cual es la función o el objetivo de la variable, de esta forma cuando veas tu código puedas entender más rápido de que se trata, y como extra escríbelas en inglés, ya que será común ver más código escritos en inglés que en español.

Ejemplos de variables correctamente nombradas en C#:

- name
- age
- error500
- _some

Como conclusión, las variables es una funcionalidad que tienen los lenguajes de programación para poder almacenar información con la cual trabajaremos, esta información debe tener un tipo, el cual definirá qué se puede almacenar, si definimos que será un número entero no podemos guardar una cadena de texto, y viceversa.

Tipos de Dato

Un tipo de datos en los lenguajes de programación es una característica que forma parte de las variables, esta característica le obligará a solo poder almacenar un conjunto de valores compatibles.

El tipo de dato es una clasificación que toman las variables, de esta forma una variable que es del tipo numérica no puede almacenar una cadena de texto, de esta manera protegemos la información y también tenemos un orden en las operaciones que se pueden realizar con las variables dependiendo su tipo.

En C# tenemos un conjunto de tipos de datos ya definidos, y hay tipos base que son llamados **tipos de datos integrados**, son tipos

de datos nativos con los cuales vamos a almacenar información estandarizada, muchos de estos tipos los verás en otros lenguajes de programación.

Hay 2 tipos de datos integrados en C#:

- Tipos de datos integrados de valor
- Tipos de datos integrados de referencia.

Los tipos integrados de valor en C# son los siguientes (no importa si no te los aprendes ahora mismo, he puesto en negritas los que considero más importantes, con el tiempo se te hará común irlos conociendo todos):

| Tipo de dato | Descripción |
|------------------------------|---|
| bool | Guarda datos booleanos, por ejemplo, si algo es verdadero o falso |
| byte | Sirve para almacenar un byte del rango 0 a 255 |
| sbyte | Sirve para almacenar un byte del rango -128 a 127 |
| char | Sirve para almacenar una letra |

| | |
|-----------------------|---|
| <u>decimal</u> | Forma parte de los tipos numéricos de punto flotante y su rango de almacenamiento es de 6 a 9 dígitos aproximadamente |
| <u>double</u> | Forma parte de los tipos numéricos de punto flotante y su rango de almacenamiento es de 15 a 17 dígitos aproximadamente |
| <u>float</u> | Forma parte de los tipos numéricos de punto flotante y su rango de almacenamiento es de 28 a 29 dígitos aproximadamente |
| <u>int</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de -2,147,483,648 a 2,147,483,647 |

| | |
|------------------------------|--|
| <u>uint</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de 0 a 4,294,967,295 |
| <u>nint</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento se calcula dependiendo la plataforma donde se utiliza |
| <u>nuint</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento se calcula dependiendo la plataforma donde se utiliza |
| <u>long</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de -9,223,378,036,854,775,808 a 9,223,372,036,854,775,807 |

| | |
|-------------------------------|--|
| <u>ulong</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de 0 a 18,446,744,073,709,551,615 |
| <u>short</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de -32,768 a 32767 |
| <u>ushort</u> | Forma parte de los tipos numéricos enteros y su rango de almacenamiento es de 0 a 65,535 |

Lo primero que te puede abrumar es que son muchos tipos base, y parece que algunos hacen lo mismo que otros, pero regularmente vas a utilizar casi siempre los mismos tipos, tu selección por algunos tipos de datos poco utilizados va a depender de una situación extraordinaria, por ejemplo, para guardar números enteros casi siempre vas a utilizar el tipo **int**, rara vez necesitar utilizar el tipo **long**, ya que rara vez estarás utilizando números así de enormes (ver en la tabla anterior el rango).

El saber cuál tipo seleccionar es parte fundamental en la programación, ya que hará que tus programas actúen de una manera esperada, si necesitas realizar operaciones con decimales, utilizarás el tipo de dato **decimal**, pero si necesitas números enormes ya tomarías la decisión de utilizar quizá el tipo **double**.

Los tipos de datos integrados de referencia solo son 3 en C#, y son tipos de datos con los cuales vamos a adentrarnos en lecciones completas más adelante.

A continuación, te muestro cuales son estos tipos de dato:

| Tipo de dato | Descripción |
|--------------|--|
| object | Es un tipo de dato del cual heredan todos los tipos de referencia (más adelante veremos que significa herencia y referencia) |
| string | Es un tipo de dato que sirve para almacenar una cadena de texto |
| dynamic | Es un tipo de datos que resuelve su tipo hasta la ejecución, permitiendo programación dinámica |

Como vimos en la lección pasada, las variables permiten guardar valores, y las variables que declaramos tienen una palabra al inicio que define su tipo.

Puedes utilizar los tipos antes vistos para crear variables las cuales deben respetar las características del tipo de dato otorgado en su creación.

A continuación, te muestro ejemplos utilizando distintos tipos y valores que son permitidos:

```
string student = "Héctor de León";  
int age = 35;  
char initial = 'H';  
  
// al utilizar tipo decimal el número debe llevar  
// una letra m al final  
decimal price = 100.12m;
```

Estructuras de control

Las estructuras de control son palabras con funcionalidad en la programación para controlar el flujo de lo que se está ejecutando.

Cuando escribimos código, cada línea de este va definiendo un flujo el cual la ejecución del programa que estamos creando va a seguir, el flujo es de arriba hacia abajo normalmente, pero con las estructuras de control podemos controlar este flujo de ejecución para que haga algo si se cumple alguna condición, o para que repita operaciones por un número de veces deseado.

Las estructuras de control principales son **estructuras de control condicionales**, las cuales nos apoyaran para realizar una operación solo si se cumple una condición, y también tenemos las **estructuras de control de bucle** con las cuales podemos repetir la ejecución de una operación.

Sentencia if y else

En la programación se menciona mucho que se basa en lógica; la lógica es una rama de la filosofía que se dedica en parte a estudiar los argumentos de que algo sea válido o no.

Las sentencias condicionantes son de las cosas más utilizadas a la hora de programar, ya que siempre hay que evaluar condiciones, para si estas se cumplen o no, hagamos alguna tarea en específico.

La sentencia If es una estructura de control condicional, ya que hará que se ejecute una o varias líneas de código sólo si se cumple una condición en específico.

Por ejemplo, si tú tienes antojo de una cerveza, la condición para poder beber una cerveza debería ser que tengas dinero y un lugar (legal) donde tomarla, solo si se cumplen estas condiciones podrás hacer esa tarea.

Gracias a la sentencia If podemos evaluar si un criterio es válido. Por ejemplo, evaluar si una edad guardada en una variable es mayor de edad (En México la mayoría de edad es a los 18 años).

```
int age = 20;  
  
// La sentencia If evalúa si una condición  
// es válida  
if (age >= 18){  
    Console.WriteLine("Mayor de edad");  
}
```

Como podemos ver en el ejemplo anterior, la sintaxis de una sentencia if contiene la palabra if y posterior paréntesis de apertura y cerradura "()", en los cuales internamente se tiene una condición a evaluar si es verdadera o falsa.

La sentencia if sirve para evaluar condiciones, si estas son válidas se hace algo, y ese algo está dentro de llave de apertura y cerradura "{ código aquí; }".

Nota: Las llaves de apertura y cerradura por ejemplo { aquí va código }, sirven para agrupar código, en las estructuras de control nos ayudarán a agrupar un bloque de código el cual se ejecutará como una unidad.

Cuando ejecutamos la instrucción anterior obtenemos como resultado:

Mayor de edad

Ya que en nuestra variable **age** tenemos un valor 20 el cual es mayor a el número 18.

Otra cosa que puedes observar es el operador **>=** el cual indica una operación de comparación, la operación de si lo que está a la izquierda es mayor o igual que lo que está a la derecha, en caso de que sea cierto esta instrucción regresa verdadero, por lo contrario regresa falso.

La sentencia **if** también tiene otros usos, como poder definir el flujo que tendrá nuestro programa cuando sea lo contrario a las condiciones evaluadas.

Para definir lo contrario a lo establecido dentro de la sentencia **if**, debemos utilizar la sentencia **else**, que significa lo contrario a lo que estamos evaluando.

Por ejemplo:

```
int age = 20;
if (age >= 18){
    Console.WriteLine("Mayor de edad");
}else{
    Console.WriteLine("No es mayor de edad");
}
```

En el código anterior tendríamos la posibilidad de evaluar lo que sea lo contrario a nuestra operación de evaluación dentro del **if**, dándonos la posibilidad de hacer algo negativo a lo que estamos validando. La instrucción **else** es un mecanismo opcional, no obligado a la hora de escribir la sentencia **if**.

Hay un tercer mecanismo que tiene la estructura de control **if** y es el **else if** el cual funciona de manera secuencial cuando la condición del **if** no se ha cumplido tal cual la sentencia **else** pero añadiendo una condición extra, esto para que se realice una tarea cuando no se cumpla la primer condición pero solo si se cumple una segunda condición.

Por ejemplo:

```
int age = 14;  
  
if (age >= 18){  
    Console.WriteLine("Mayor de edad");  
}else if (age >= 13) {  
    Console.WriteLine("Es adolescente");  
}  
else {  
    Console.WriteLine("Es un niño");  
}
```

Al ejecutar este código tendríamos como resultado:

Es un adolescente

En el ejemplo anterior podemos observar que evaluamos la condición de una edad de 14 almacenada en la variable `age`, en la comparación evaluamos si es mayor o igual a 18 por lo cual no ingresa a realizar lo que está interno, pasando a la siguiente condición **else if** en la cual se evalúa si es mayor a 13 años cumpliendo con esto, por lo cual el mensaje arrojado es que "es un adolescente".

Si nuestra variable tuviera un valor menor a 13, ingresaría al último **else**, el cual se ejecutaría cuando no se cumpla ninguna condición anterior.

Podemos utilizar tantos **else if** como deseemos, aunque más adelante veremos estructuras de control más aptas para evaluar múltiples condiciones, pero antes vamos a ver que son los operadores de comparación.

Operadores de comparación

Los operadores de comparación nos sirven para comparar dos valores e indicarnos si la comparación es verdadera o falsa.

En el ejemplo anterior vimos el operador que evaluaba si un valor es mayor o igual a otro (**>=**), en caso de que fuera verdad se nos otorga verdadero, o mejor dicho en programación un valor booleano **true** (a partir de ahora indicaremos todo valor booleano en inglés: **true o false**).

Nota: Cuando me refiera a valor booleano o boolean es un valor el cual puede ser true o false.

Los valores booleanos es un valor que nos indica si algo es true o false, y con los operadores de comparación podremos evaluar valores.

Estos operadores se utilizan junto a las estructuras de control.

Los operadores para comparar más utilizados son:

| Operador | Descripción |
|-----------|---------------------|
| == | Valores son iguales |
| != | Valores distintos |

| | |
|----|--|
| < | Evalúa si el valor de la izquierda es menor que el de la derecha |
| > | Evalúa si el valor de la izquierda es mayor que el de la derecha |
| <= | Evalúa si el valor de la izquierda es menor o igual que el de la derecha |
| >= | Evalúa si el valor de la izquierda es mayor o igual que el de la derecha |

Estaremos utilizando estos operadores a lo largo de todo este libro.

Operadores lógicos booleanos

Los operadores lógicos booleanos sirven para trabajar con operaciones lógicas, ya sea para enlazar varias evaluaciones de condiciones o para negar toda una comparativa.

Algunas veces necesitaremos evaluar varias cosas, por ejemplo para tomar cerveza tu necesitas tener dinero y ganas de tomar cerveza, son dos condiciones, y para este tipo de comparativas necesitamos de estos operadores.

Los operadores principales son:

| Operador | Descripción |
|----------|---|
| && | indica que debe cumplirse lo que está a la izquierda y lo que está a la derecha |
| | Indica que debe cumplirse lo que está a la izquierda o lo que está a la derecha |
| ! | Le da negación completa a una condición que esté a la derecha |

Estos operadores los estarás utilizando todos los días como programador.

En el siguiente ejemplo evaluamos utilizando estos operadores:

```
bool wannaDrink = false;
int money = 150;

if (money > 0 && wannaDrink)
{
    Console.WriteLine("Se toma una cerveza");
}
```

En el ejemplo anterior evaluamos si el dinero es mayor a 0 y si tenemos ganas de tomar, si vemos la variable **wannaDrink** por sí

misma es un tipo boolean, es decir, ya por sí misma tiene un valor lógico que puede tomar true o false.

El operador de negación funciona de la siguiente forma:

```
bool isDead = false;  
if (!isDead)  
{  
    Console.WriteLine("Está vivo");  
}
```

El operador de negación niega lo que esté a su derecha, por lo cual el código anterior imprime en pantalla: "Está vivo".

Sentencia Switch

Hay ocasiones donde necesitamos evaluar múltiples escenarios, y hacerlo con la sentencia **if** nos resultará fastidioso.

La sentencia switch nos permite evaluar múltiples condiciones de una manera más organizada, haciendo que escribamos código más entendible.

A continuación, podemos ver un ejemplo utilizando la sentencia **switch**:

```
int age = 62;  
  
// Ponemos el valor a evaluar entre paréntesis  
  
switch (age)  
{  
    // cada caso irá evaluando iniciando con la palabra case  
    // y es necesario se cierre con la instrucción break
```

```
case < 3:
    Console.WriteLine("Es un bebé");
    break;
case < 13:
    Console.WriteLine("Es un niño");
    break;
case < 18:
    Console.WriteLine("Es adolescente");
    break;
case < 60:
    Console.WriteLine("Es mayor de edad");
    break;
case < 120:
    Console.WriteLine("Es de la tercera edad");
    break;

// podemos poner una opción final por si no ingresa
// a ninguna anterior con la palabra default la definimos
default:
    Console.WriteLine("Valor no válido");
    break;
}
```

Al ejecutar el código anterior tendríamos como resultado:

Es de la tercera edad

Este resultado es obtenido ya que nuestra variable tiene un valor de 62, y ese valor solo cumple con la opción donde es menor a 120 pero no cumple con las opciones antes evaluadas, recuerda que el código se va ejecutando y evaluando de arriba hacia abajo.

La sentencia switch no sirve para cuando necesitemos evaluar un valor en distintos escenarios, por ejemplo, evaluar rangos como en

el código anterior que evaluamos una edad dentro de un rango que la describe.

Te recomiendo no utilices switch en casos donde tengas evaluaciones mayores a 5 casos, ya que se vuelve mucho código y es difícil darle mantenimiento, sobre todo cuando ponemos mucho código dentro de cada caso; más adelante en la sección de **array** te mostraré otras formas para tener código más limpio en situaciones donde se superen los 5 casos en un switch. Pero no está demás conocer esta sentencia.

Bucle While

La primera **estructura de control de bucle** que veremos es **while**, la cual significa “**mientras**” en español.

Las estructuras de control de tipo bucle sirven para repetir secciones de código.

En la programación es necesario algunas veces estar realizando operaciones varias veces, por ejemplo si necesitamos crear un programa que genere un archivo con una estructura fija para cada persona registrada en un sitio web, este archivo siempre se generará utilizando el mismo código de programación solo cambiando alguna información (la información del usuario), pero el funcionamiento general para crear el archivo es el mismo, en este tipo de situaciones es útil las estructuras de control de tipo bucle.

Otra situación es cuando creamos en un sitio web la sección de noticias, el contenido HTML de la sección de noticias es el mismo para cada noticia, se repite el mismo cuerpo de código, en este tipo de situación también es útil este tipo de estructuras de control.

Esta instrucción ejecuta un bloque de código siempre y cuando se cumpla una condición, esta condición evalúa si es verdadero o falso,

cuando sea verdadero se repetirá el código interno en la sentencia **while**.

```
int counter = 0;
int times = 5;

// Se repetirá hasta que counter llegue al valor de times

while (counter < times)
{
    // Al poner una cadena de texto la cual esta entre comillas
    // podemos concatenar un valor con el símbolo: +

    Console.WriteLine("Se ejecuta: " + counter);

    // incrementamos la variable counter
    counter = counter + 1;
}
```

Tendríamos el siguiente resultado en pantalla:

```
Se ejecuta: 0
Se ejecuta: 1
Se ejecuta: 2
Se ejecuta: 3
Se ejecuta: 4
```

Bucle Do While

La sentencia **Do While** también forma parte de las estructuras de control de bucle.

La diferencia de Do While respecto a While es que por lo menos se ejecutará una vez y posterior a esta ejecución evaluará si repetirá el código que le corresponde.

A continuación, vemos un ejemplo de su funcionamiento:

```
int counter = 0;
int times = 5;

// ingresa sin necesidad de evaluarse

do
{
    Console.WriteLine("Se ejecuta: " + counter);

    // podemos utilizar el operador ++, esto incrementará
    // el valor de la variable + 1
    counter++;

    // en este punto se evalúa si se repetirá el código de arriba
    // nuevamente caso contrario sale de este bucle la ejecución.
    // Observa que al final de while() hay un punto y coma (;)
}while (counter < times);
```

El resultado obtenido sería el siguiente:

```
Se ejecuta: 0
Se ejecuta: 1
Se ejecuta: 2
Se ejecuta: 3
Se ejecuta: 4
```

En este ejemplo obtendremos el mismo resultado que con la instrucción while, pero toma en cuenta que el código se ejecuta por lo menos 1 vez, y posterior se evalúa si se repetirá.

Do While es útil para situaciones donde necesitamos realizar una tarea y si esta falla por alguna situación poder reintentar nuevamente.

Bucle For

La sentencia **For** es parte de las **estructuras de control de bucle**, y nos servirá para tener mejor manejo de las iteraciones de nuestro código.

Su funcionamiento se centra en 3 partes:

- La primera parte es donde iniciamos los valores
- La segunda parte es una condición que mientras siga siendo true seguirá repitiendo el código interno.
- Y la tercera parte es donde indicaremos que haremos al final de la ejecución de nuestro código interno (regularmente ponemos un incremento a una variable para que continúe hasta llegar al fin deseado).

```
// for consta de 3 partes separadas por punto y coma (;)  
// En la primera iniciamos una variable i con valor 0  
// En la segunda comparamos si i es menor a 10  
// En la tercera parte incrementamos i  
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine("Ejecución número: " + i);  
}
```

El resultado sería el siguiente:

```
Ejecución número: 0  
Ejecución número: 1  
Ejecución número: 2  
Ejecución número: 3  
Ejecución número: 4  
Ejecución número: 5
```

| |
|--|
| Ejecución número: 6 Ejecución número: 7 Ejecución número: 8 Ejecución número: 9 |
|--|

Como podemos observar, con la sentencia `for` podemos tener control de las repeticiones de la ejecución de nuestro código creando e incrementando los valores dentro de la misma estructura de la estructura de control.

Comparando las 3 estructuras de control de bucle que hemos visto hasta este punto, yo te recomiendo utilizar:

- Sentencia **while**: cuando no conozcas el número de repeticiones deseado y lo tengas que resolver internamente en el código de esta estructura, por ejemplo, cuando se ejecuta algo y deseas reintentar, pero que para esta ejecución se deba cumplir una condición inicial.
- Sentencia **do while**: mismo caso que `while` con la diferencia de que necesites ejecutar algo por lo menos 1 vez, y posterior quizá repetir la ejecución dependiendo la situación.
- Sentencia `for`: cuando conozcas el número de repeticiones a realizar, o cuando utilices arreglos (los veremos en la siguiente sección).

Funciones

Las funciones son un concepto base de la programación que existe en todos los lenguajes de programación más utilizados.

Una función es un bloque de código el cual realiza una tarea en específico.

Este bloque de código tiene un nombre, puede tener parámetros de entrada y puede tener un valor de retorno.

Imagina que necesitas calcular un impuesto y la fórmula siempre es la misma, estar calculando el impuesto en distintas partes de tu

código puede volverse tedioso, sobre todo si el impuesto un día cambia en la forma de cómo se calcula.

Otro ejemplo de función es tener el funcionamiento para conectarnos a una base de datos, esta conexión siempre es el mismo proceso, el tener una función con este proceso nos ayuda a no tener que repetir el código cada que necesitemos de esto, evitando estar escribiendo una y otra vez el mismo código.

Una función te sirve para encapsular el código necesario para realizar una tarea, dándole nombre y poder ser llamada cuando la necesites.

También una función nos sirve para tener un funcionamiento en específico en un solo sitio, así si necesitamos modificar dicho funcionamiento, solo tenemos que ir a un sitio y modificarlo.

A continuación, te muestro un ejemplo de una función

```
// Este es el ejemplo de una función que muestra en pantalla un  
// texto  
// void indica que nuestra función no regresa ningún valor  
void Hi()  
{  
    Console.WriteLine("iHola, mundo!");  
}  
  
// Para ejecutar su código basta con que escribamos su nombre  
// y los paréntesis al final ();  
Hi();  
Hi();  
Hi();
```

El resultado sería el siguiente

iHola, mundo!


```
iHola, mundo!  
iHola, mundo!
```

Hemos llamado la función 3 veces, y esta ejecuta el código que tiene internamente.

También una función puede recibir parámetros:

```
// La función Hi ahora puede recibir un nombre y un apellido  
// A los parámetros se les indica un tipo, son variables  
  
void Hi(string name, string lastName)  
{  
    // El operador + cuando se utiliza string concatena el texto  
    Console.WriteLine("iHola, " + name + " " + lastName + "!");  
}  
  
// Se debe mandar el valor del tipo que espera la función  
Hi("Héctor", "De León");
```

Y el resultado sería el siguiente:

```
iHola, Héctor De León!
```

Además, las funciones tienen la característica de poder regresar un valor:

```
// Función que recibe dos parámetros y regresa un valor int  
// int al inicio indica que nuestra función regresa un valor int  
  
int Add(int num1, int num2) {
```

```
// Se realiza la devolución de un valor con la palabra return  
// En este caso el operador + sirve para sumar 2 números  
return num1 + num2;  
}  
  
// Ejecutamos la función Add enviándole 2 valores y guardando  
// lo que nos retorna en la variable res la cual es int  
  
int res = Add(5, 8);  
Console.WriteLine(res);
```

Nuestra pantalla debería mostrar lo siguiente:

13

En la sección de programación funcional veremos más cosas que se pueden hacer con las funciones.

Las funciones son una herramienta que vas a utilizar todos los días en tu trabajo como programador.

3. Array

¿Qué es un array?

Un **array** es una **estructura de datos** que nos permite guardar múltiples valores del mismo tipo.

Nota: *Las estructuras de datos en programación son mecanismos que nos permiten organizar información, un array es una estructura de datos.*

Un array es una variable la cual puede tener una colección de elementos del mismo tipo.

Los array manejan sus elementos de forma secuencial asignándoles una posición que comienza con el número 0.

```
// Un array se define por el tipo al inicio, en este caso string
// Seguido de dos corchetes []
// Luego sigue el nombre, es una variable
// Al final la palabra new seguido del tipo y entre corchetes
// su longitud [5], este array puede almacenar 5 elementos

string[] beers = new string[5];

// Para asignar valor indicamos la posición o mejor dicho índice
// Los array comienzan en 0
beers[0] = "Erdinger";
beers[1] = "Delirium";
beers[2] = "Minerva";
beers[3] = "Corona";
beers[4] = "Paulaner";

Console.WriteLine("La primer cerveza es: "+ beers[0]);
```

En nuestra pantalla tendríamos el siguiente resultado:

La primer cerveza es: Erdinger

Nota: La posición en un arreglo es llamada **índice**, y esta debe ser un número entero, puedes utilizar un valor o una variable, pero debe ser del tipo de dato `int`.

Recorrido de un array

Un array puede almacenar una colección de elementos, y con lo aprendido hasta este punto podemos recorrer la información que contiene para realizar alguna acción con ella.

Para poder recorrer un array podemos utilizar una estructura de control de bucle, y en este caso nos serviría utilizar la estructura `for`.

```
string[] names = new string[3];

names[0] = "Héctor";
names[1] = "Pedro";
names[2] = "Ana";

// Iniciamos una variable i en 0 y la comparamos con la longitud //
// del array
// Un array tiene una propiedad Length que nos regresa su tamaño
// Incrementamos i++ hasta que sea igual al tamaño del array
// Recordemos que los array comienzan en 0

for (int i = 0; i < names.Length; i++)
{
    // Para mostrar la información ponemos la variable i en []
    // La variable i se irá incrementando en cada iteración

    Console.WriteLine(names[i]);
}
```

El resultado sería el siguiente

| |
|------------------------|
| Héctor Pedro Ana |
|------------------------|

Funciones que regresan un array

Las funciones pueden regresar un valor por lo cual pueden regresar un array como resultado.

Por ejemplo, podemos crear una función que nos llene un array, la cual reciba la cantidad de elementos que deseamos y que el array ya incluya los números en secuencia en un array como resultado.

```
// Definimos una función Generate que regresa un array int[]
// Esta función recibe como parámetro la longitud que deseamos
int[] Generate(int length)
{
    // Creamos un array interno con la longitud recibida
    int[] numbers = new int[length];

    // Repetimos hasta llegar a la longitud recibida
    for (int i = 0; i < length; i++)
    {
        // Vamos llenando el array con el valor que tiene i + 1
        // El resultado de i + 1 se guarda en el array

        numbers[i] = i + 1;
    }
    // Regresamos el array con los valores
    return numbers;
}

// Ejecutamos la función y tendremos un array ya con valores
int[] numbers = Generate(10);

// Podemos recorrer el array generado
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine(numbers[i];)
}
```

Y el resultado sería:

```
1
2
3
```

4

5

6

7

8

9

10

Alternativa a Switch, utilizando Array

En la sección de la sentencia Switch se mencionó que cuando se tienen muchas opciones utilizar Switch no puede ser la mejor solución, sobre todo si nuestro código tiene posibilidad de aumentar opciones.

No necesariamente siempre es mejor utilizar array, pero hay situaciones donde se reducirá el código bastante, y será más fácil hacer cambios en nuestro programa si se requiere agregar nuevas opciones.

Por ejemplo, cuando estamos utilizando la sentencia Switch para asignar un valor a una variable dependiendo la situación como nuestro a continuación:

```
string style = "";  
int op = 1;  
  
// Se asigna a style dependiendo el valor de op  
switch (op)  
{  
    case 1:  
        style = "Stout";  
        break;  
    case 2:  
        style = "Porter";  
        break;  
    case 3:  
        style = "Pale Ale";  
        break;  
    case 4:  
        style = "APA";  
        break;  
    default:  
        style = "Sin estilo";
```



```
break;  
}
```

Con un array podríamos tener el mismo comportamiento de la siguiente manera:

```
string[] styles = new string[4] {  
    "Stout", "Porter", "Pale Ale", "APA"  
};  
int op = 1;  
  
// los array comienzan en posición 0, por eso restamos 1  
style = styles[op - 1];
```

El código anterior nos permite tener mayor control a la hora de agregar opciones, solo bastaría con ir a agregar un nuevo valor al array.

En la programación tenemos distintas alternativas, y dependiendo la situación será más práctico utilizar algo respecto a otra solución, con la experiencia que iremos ganando sabremos qué es lo más viable dependiendo el escenario del problema.

4. Programación Orientada a Objetos Básica

En esta sección vamos a abordar elementos básicos de la programación orientada a objetos.

He dividido este tema en 2 secciones siendo esta la sección donde veremos los conceptos más básicos y fundamentales.

Más adelante se encuentra la sección de programación orientada a objetos avanzada, donde abordaremos conceptos no tan básicos, esto posterior a ver otros temas con los cuales entenderás de una manera más clara.

Paradigmas en la programación

Un paradigma en programación es una manera de resolver problemas.

Un paradigma te proporcionará un grupo de conceptos, principios y prácticas para programar.

Los ejemplos que hemos hecho hasta este punto en este libro se han hecho con el paradigma estructurado, un paradigma de programación el cual se ejecuta de manera secuencial y se apoya de las estructuras de control.

El paradigma estructurado es una manera de resolver problemas, pero tiende a ser problemático ya que no tiene buena organización.

En esta sección nos enfocaremos en el paradigma de programación Orientado a Objetos, el paradigma de programación más utilizado.

¿Qué es la Programación Orientada a Objetos?

La programación orientada a objetos es un paradigma de programación el cual se centra en la estructura de datos llamada: **Objeto**.

Los objetos son entidades que guardan información y pueden realizar acciones.

Podemos ver que en nuestra vida diaria estamos rodeados de objetos, de hecho, nosotros mismos somos un objeto.

Por ejemplo, un auto es un objeto tiene comportamientos ya establecidos, sabemos que puede caminar, también que pueda dar vuelta, o frenar. Un auto también tiene propiedades que lo definen: llantas, puertas, color, velocidad máxima.

Con la programación Orientada a Objetos vamos a organizar nuestro código en objetos, los cuales tendrán la funcionalidad y propósito de resolver un problema.

Clases y Objetos

En la programación orientada a objetos trabajaremos con la entidad objeto, un objeto tiene comportamiento el cual puede realizar y características para almacenar información.

La definición de lo que tiene un objeto y puede hacer se realiza en una clase.

```
class Sale{  
    // Propiedades que puede tener el objeto  
    // Por ahora ignora la palabra public, más adelante explicaremos  
    public decimal Total;  
    public string Customer;
```

```

// Comportamiento que puede tener el objeto
public string GetInfo(){
    // creamos un texto informativo
    string info = "El Total es $ " + Total;

    // el operador += en cadenas sirve para concatenar
    info += " y el cliente es: " + Customer;

    // regresamos el valor que está en info
    return info;
}
}

```

En el código anterior podemos observar que nuestra clase Sale, definida con la palabra **class** tiene 2 propiedades: una llamada Total que es del tipo de dato decimal, y otra Customer del tipo de dato string, las propiedades son variables tal cual hemos visto anteriormente.

Una clase es una plantilla de qué es lo que puede tener un objeto y que es lo que puede hacer.

También nuestra clase tiene un comportamiento definido con una función llamada GetInfo con la cual obtenemos la información de la venta.

En una clase tenemos variables que representan las propiedades o características que puede tener un objeto y funciones que representan su comportamiento.

Para hacer uso de esta clase definida por nosotros basta con hacer lo siguiente:

```

// Nombre de nuestra clase seguido del nombre de nuestro objeto
// seguido de la palabra new y posterior el nombre de la clase y //
// paréntesis ()
Sale sale = new Sale();

// asignamos los valores de las propiedades

```

```
sale.Total = 100;  
sale.Customer = "Pablo";  
  
// ejecutamos el método GetInfo()  
string info = sale.GetInfo();  
Console.WriteLine(info);
```

Y en nuestra pantalla tendremos el siguiente resultado:

```
El Total es $ 100 y el cliente es: Pablo
```

En esta sección profundizaremos más en las clases y los objetos.

Constructor

El constructor es un método especial, el cual se invocará al momento que creamos un objeto.

El objetivo del constructor es inicializar valores que son necesarios en el objeto.

Un constructor tiene el mismo nombre de la clase y no regresa ningún valor, pero si puede recibir parámetros.

El constructor se invocará en automático al momento que se crea un objeto, esto ocurre al momento que utilizamos la palabra **new** más el nombre de nuestra clase.

```
// Aquí se invoca en automático el constructor  
MyClass object = new MyClass();
```

Para crear un constructor tenemos que hacer lo siguiente:

```
class Sale  
{  
    // Propiedades que puede tener el objeto  
    public decimal Total;  
    public string Customer;  
  
    // constructor debe llamarse igual a la clase y no retornar nada  
    public Sale(decimal total, string customer)  
    {  
        // se asignan los valores recibidos a las propiedades de  
        // la clase  
  
        Total = total;  
        Customer = customer;  
    }  
}
```

En el caso anterior en la clase Sale, para construir un objeto, nos obliga el constructor a enviar 2 valores, y tendríamos que hacerlo de la siguiente manera:

```
// Creación de objeto con constructor que obliga a enviar valores  
Sale sale = new Sale(16, "Héctor");
```

De esta manera nuestra clase nos obliga a enviar los valores que son necesarios para la existencia del objeto.

Cuando no creamos un **constructor** en nuestra clase, ya se tiene un **constructor** por defecto implícitamente el cual no recibe parámetros.

Herencia

La herencia es una característica de la programación orientada a objetos que nos permite reutilizar funcionalidad de una clase ya existente.

La clase de la cual heredamos la llamaremos clase padre y la clase que hereda de otra la llamaremos clase hija.

En sí, esta característica es poder reutilizar y extender algo ya existente, para no tener que estar escribiendo código repetitivo.

A continuación, un ejemplo de cómo se realiza la herencia:

```
// Clase padre  
class Water{  
    public string Name;  
}  
  
// Clase hija  
// Para realizar la herencia debemos escribir dos puntos(:)  
// seguido del nombre de la clase padre  
class Beer : Water{  
    public string Style;
```

```
}
```

En el ejemplo anterior tenemos una clase llamada Water la cual tiene una propiedad llamada Name. Y también tenemos una clase hija llamada Beer la cual hereda de Water y esta última tiene una propiedad extra llamada Style.

Gracias a la herencia los objetos creados con la clase Beer tienen la propiedad Name, y a parte tiene una propiedad extra llamada Style, por lo cual los objetos pueden utilizar Name y Style.

```
// Creación del objeto Beer  
Beer myBeer = new Beer();  
  
// Name: Propiedad de la clase padre obtenida por herencia  
myBeer.Name = "Cerveza Fuerte";  
  
// Style: propiedad existente solo en la clase Beer  
myBeer.Style = "Stout";
```

En C# una clase sólo puede heredar de otra, no puedes heredar de varias clases, es decir, no puedes hacer multi-herencia de 2 padres. Lo que sí puedes hacer es tener una clase abuela (utilizando analogía de jerarquía) a la cual hereda una clase padre, de la cual hereda una clase hija.

Esta característica está relacionada a algunas otras de la programación orientada a objetos que veremos más adelante, y estaremos viendo más ejemplos de herencia.

Constructores en Herencia

El constructor en una clase es un método que nos obliga a enviar en la creación del objeto parámetros, en el caso de la herencia, la clase hija está obligada a enviar parámetros al constructor del padre siempre y cuando exista.

Para que nuestra clase cumpla con ello vamos a enviar el valor o valores necesarios a la clase padre de la siguiente manera:

```
// Clase padre
class Water{
    public string Name;

    // Constructor de clase padre
    public Water(string name)
    {
        Name = name;
    }
}

// Clase Hija
class Beer : Water{
    public string Style;

    // Constructor clase hija
    // Para enviar el valor obligado por clase padre se envía
    // utilizando el método base el cual recibirá los mismos
    // parámetros que la clase padre, en este caso name
    public Beer(string style, string name) :
        base(name)
    {
        Style = style;
    }
}
```

La clase padre en el ejemplo anterior necesita de un parámetro llamado `name` que es `string` para poder crear un objeto, desde la clase hija enviamos el valor por medio del método **base**, el cual recibirá el mismo número de parámetros que tiene el constructor de la clase padre.

De esta manera, para crear un objeto de la clase `Beer` tendríamos que hacer lo siguiente:

```
Beer myBeer = new Beer("Stout", "Cerveza fuerte");
```

Encapsulamiento

El encapsulamiento es una de las cualidades de la programación orientada a objetos que sirve para ocultar y proteger la información y el comportamiento de los objetos.

Gracias a esta característica podemos hacer funcionalidad protegida, y también información protegida.

Por ejemplo, un objeto que representa un auto, lo único que nos interesa para conducirlo es saber cómo enciende, camina, frena y darle dirección, pero ¿Qué pasaría si modificamos el mecanismo con el cual funciona el motor sin saber de mecánica? Tenemos un alto riesgo de hacerlo inservible.

Igual en la programación, los objetos pueden tener mecanismos internos e información la cual no debería ser alterada cuando este se está utilizando.

El encapsulamiento nos ayuda a indicarle a la clase que se puede utilizar en la misma clase y que no se puede utilizar fuera de la clase.

La palabra reservada **public** que hemos estado viendo en los ejemplos anteriores es parte del mecanismo del encapsulamiento, **public** indica que lo definido ya sea método o ya sea propiedad puede ser utilizado fuera de la clase.

Lo contrario a public existe como algo que es privado y es categorizado con la palabra reservada private, de esta manera protegemos que la propiedad o el método sólo pueda ser utilizado dentro de la misma clase donde es codificado.

A continuación, te muestro un ejemplo:

```
// Clase impuesto
class Tax{
    // propiedad pública
    public decimal Amount;

    // 1. Campo privado, no puede accederse fuera de la clase
    // 2. He puesto un guión bajo en el nombre ya que es privado
    // esto es parte de las convenciones(sugerencias para
    nombrar)
    // de nombramiento en C#
    private decimal _rate;

    // El objeto recibe al crearse el monto y la tasa de impuesto
    public Tax(decimal amount, decimal rate){
        // Se asignan los valores
        Amount = amount;
        _rate = rate;
    }

    // Método público que puede accederse por fuera de la clase
    public decimal GetTotal(){
        return Amount * _rate;
    }
}
```

Y para utilizarse el siguiente objeto es así:

```
Tax tax = new Tax(10m, 1.16m);

// Se permite invocar el método GetTotal() porque es público
Console.WriteLine(tax.GetTotal());
```

```
// Amount es público  
Console.WriteLine(tax.Amount);  
// No se puede acceder a Rate ya que es privado  
Console.WriteLine(tax._rate);
```

El encapsulamiento nos permite tener una programación protegida para no alterar funcionamiento o información que altere el objetivo de la programación del objeto que estamos creando. Es decir, nos da seguridad para permitir que puede ser modificado fuera de la codificación de la clase y que no puede ser alterado sin que se rompa el objetivo de lo que el objeto debería realizar.

Encapsulamiento en Herencia

El encapsulamiento también tiene un papel importante al momento de utilizar herencia en nuestro código.

Anteriormente vimos la palabra reservada **public** que sirve para hacer un elemento accesible en cualquier lado y la palabra reservada **private** que sirve para hacer un recurso sólo accesible dentro de la misma clase, pero existen otras palabras reservadas con las cuales vamos a ver su funcionamiento al momento de codificar herencia.

Podemos hacer que un método o propiedad de una clase padre sea accesible sólo para sus clases hijas existe la palabra reservada **protected**.

```
// Clase padre con elementos protected  
class People{  
    // campo protegido, solo pueden usarse por este clase y clases  
    // hijas que hagan herencia de esta clase.  
    protected string name;  
  
    public People(string name){  
        // cuando tenemos campos con el mismo nombre, this  
sirve  
        // para referirnos al campo de esta misma clase y no al  
        // recibido como parametro
```

```

        this.name = name;
    }

    // protected también sirve para los métodos
    protected string GetDescription(){
        return "El nombre es: " + name;
    }
}

```

Ahora el código de la clase hija que hereda de la clase anterior:

```

// Clase hija
class Engineer : People{
    private string _profession;

    public Engineer(string name, string profession)
        : base(name) {
        _profession = profession;
    }

    public string GetInfo(){
        // ejecutamos el método GetDescription() protegido de la
        // clase padre
        return "La profesión es: " + _profession + " " +
            GetDescription();
    }

    public string GetName(){
        // retornamos el valor del campo protegido name
        return name;
    }
}

```

En el ejemplo anterior tenemos una clase llamada **People** la cual tiene un campo protegido **name** y un método protegido **GetDescription()**, al ser protegidos podemos acceder tanto en la misma clase **People** como en clase que hereden de esta, en este caso la clase **Engineer**.

```
Engineer hector = new Engineer("Héctor", "Programación");  
Console.WriteLine(hector.GetInfo());  
  
// No podemos acceder al campo protegido name ya que es  
protected  
Console.WriteLine(hector.name);  
  
// Pero si podemos acceder utilizando el método publico GetName()  
Console.WriteLine(hector.GetName());
```

El encapsulamiento debe ser utilizado como un protector de información y comportamientos, dependiendo la situación que necesitemos vamos a optar ya sea por una manera pública, protegida o privada de nuestros elementos.

En C# existen otros operadores de encapsulamiento que tienen funcionamiento a la hora de manejar ensamblados y creación de bibliotecas de utilidades (este libro no hablaremos de esos temas), solo dejare mención de ellos aquí:

- **internal**: su alcance solo es dentro de los mismos archivo de ensamblado
- **protected internal**: es combinación de **protected** e **internal** y su alcance abarca clases del mismo ensamblado añadiendo clases que hereden de otras clases de otro ensamblado.
- **private protected**: mismo alcance que **private** con la diferencia que no puede ser utilizado por clases que hereden de otro ensamblado.

Nota: *Un ensamblado es un conjunto de código ya compilado que funcionan como un conjunto, un ejemplo son las bibliotecas que ya existen en el lenguaje C#.*

Campos y Propiedades

Las clases en C# pueden tener tanto campos como propiedades. Un campo es una variable la cual sirve para guardar un valor de nuestro

objeto, las propiedades también sirven para esto, pero tienen una característica extra que es el poder manejar la información.

Los campos y las propiedades pueden trabajar en conjunto, junto a el encapsulamiento nos ayudarán a tener información válida en nuestros objetos.

La propiedad tiene características que permiten evaluar el valor que regresarán y validar el valor que reciben, haciendo que podamos proteger la información.

Un ejemplo es tener un objeto el cual necesite guardar edades, y por medio de una propiedad podemos validar que el valor que se asigne sea mayor a 0, ya que es imposible que exista una persona con edad negativa. De esta manera podemos hacer software con mejor calidad.

A continuación, te explico con código:

```
class Sale
{
    // El campo total es privado
    private decimal total;

    // La propiedad Total actúa como medio de acceso al campo
    // privado total
    public decimal Total
    {
        // Con la palabra get especificamos lo que se regresará
        get { return total;}

        // Con la palabra set especificamos que pasará al darle un
        // valor a la propiedad
        set {
            // La palabra value nos sirve para acceder al valor otorgado
            total = value;

            // Validamos y asignamos en caso de número negativo
```

```
        if (total < 0)
        {
            total = 0;
        }
    }
}
```

Las propiedades nos brindan funcionalidad para poder regresar un valor con un formato en especial, y en el caso de asignar un valor, poder contemplar escenarios de validación o valores por defecto.

```
Sale sale = new Sale();  
  
// Si asignamos un valor a la propiedad pública Total de -1  
sale.Total = -1;  
  
// Obtendremos un valor por defecto de 0 al obtenerlo de la  
// propiedad pública Total  
Console.WriteLine(sale.Total);
```

Static

En la programación orientada a objetos existe un modificador llamado static, el cual nos servirá para definir elementos ya sean métodos, campos o propiedades las cuales se puede acceder a ellos sin necesidad de crear un objeto, simplemente utilizando la clase.

Esta característica puede ser útil para crear elementos globales, el cual va ser único y compartido entre todo funcionamiento que utilice la clase que lo contenga.

A continuación, te muestro un ejemplo de esta característica:

```
class Beer{
    // El operador static puede ir en una propiedad
    public static int Quantity = 0;
    public Beer(){
        // cuando se crea un nuevo objeto se incrementa Quantity
        Quantity++;
    }
    // También podemos aplicar que un método sea estático
    public static void Show(){
        Console.WriteLine(Quantity);
    }
}
```

El operador static nos permite crear elementos que pertenecen a la clase, a los cuales podemos acceder directamente escribiendo el nombre de la clase, sin necesidad de crear un objeto.

Al pertenecer a la clase algo, su contexto pasa a ser global, al cual podemos acceder desde cualquier parte de nuestro código.

Podemos ver un ejemplo aquí:

```
// Al crear un objeto la propiedad Quantity se incrementa
var myBeer1 = new Beer();

// En este punto tendríamos en la pantalla un 1 impreso
// Accedemos a Quantity directamente de la clase y no del objeto
Console.WriteLine(Beer.Quantity);

// Al crear 2 nuevos objetos Quantity se incrementa nuevamente
var myBeer2 = new Beer();
var myBeer3 = new Beer();

// En este punto Quantity nos daría un 3
Console.WriteLine(Beer.Quantity);
```

```
// Para acceder a un método estático, basta con escribir la clase  
// y el método estático, sin necesidad de crear un objeto.  
Beer.Show();
```

El modificador static puede ser útil en situaciones donde necesitemos compartir información o cuando tengamos métodos que realicen una tarea en específico como de realizar una fórmula matemática o el cálculo de un impuesto, pero no hay que abusar de este operador ya que puede ser difícil darle mantenimiento cuando nuestro software va escalando.

Nota: El método que imprime en pantalla y que hemos estado utilizando en los ejemplos **Console.WriteLine**, es un método estático, es por ello que no hacemos un objeto para su uso.

Sobrecarga

La sobrecarga es una característica que tiene la Programación Orientada a Objetos con la cual podemos definir múltiples métodos que se llamen igual, pero estos tengan distintos parámetros.

La distinción de parámetros va en cuanto a su tipo y la cantidad de estos en el método a sobrecargar, un método llamado **Suma(int a, int b)** puede sobrecargarse con otro llamado **Suma(string a, string b)**, y no tendríamos problema que se llame igual, ya que sus parámetros son de tipo distinto.

Esta característica nos provee flexibilidad a la hora de crear métodos, dándonos la posibilidad de poder recibir distintas fuentes de entrada y dependiendo la situación realizar el propósito del método.

```
// Esta clase tiene 3 métodos con el mismo nombre gracias a la  
// sobrecarga  
class Math{  
    // Método que suma 2 números del tipo int  
    public int Add(int x, int y)
```

```

    {
        return x + y;
    }

    // Método que suma 3 números del tipo int
    public int Add(int x, int y, int z)
    {
        return x + y + z;
    }

    // Método que suma 2 números introducidos como cadena, tipo
de
    // dato string
    public int Add(string x, string y)
    {
        // Al utilizar int.Parse() convertimos una cadena a tipo
        // de dato int
        return int.Parse(x) + int.Parse(y);
    }
}

```

En la clase Math tenemos 3 métodos llamados Add, los cuales tienen distintos parámetros de entrada, dependiendo la necesidad, ejecutamos el método deseado.

Estamos brindando distinto comportamiento a nuestro objeto, el cual dependiendo la situación invocará el método correspondiente, dependiendo los datos de entrada, siendo estos los parámetros que enviamos al método.

Podemos ver la ejecución de un objeto creado a partir de la clase Math a continuación:

```

Math math = new Math();

// Se invoca el método Add dependiendo la necesidad de datos de
// entrada
int res1 = math.Add(1, 3);

```

```
int res2 = math.Add(1, 2, 3);  
int res3 = math.Add("1", "2");
```

La sobrecarga nos dará una forma intuitiva de invocar un funcionamiento dependiendo de nuestros datos de entrada.

Sobreescritura

La sobreescritura es un concepto de la Programación Orientada a Objetos que proporciona el poder a una clase hija de reescribir un método de su clase padre.

Esto es útil cuando estamos realizando herencia, pero el método de la clase padre no nos sirve o nos sirve parcialmente, la sobreescritura sirve para estas situaciones.

```
// Clase padre  
class Sale{  
    protected decimal _total;  
    // En C# para especificar que un método puede ser sobreescrito  
    // debemos escribir la palabra virtual  
    public virtual void Add(decimal amount){  
        _total += amount;  
    }  
}
```

Teniendo una clase padre llamada **Sale**, la cual tiene un método llamado **Add** que incrementa su campo **_total**, al momento que realizamos una herencia a esta clase, con la sobreescritura podemos extender su funcionamiento.

Al heredar de la clase anterior reutilizamos y asignamos nuevo funcionamiento:

```
// Clase hija
class SaleWithTax : Sale
{
    private decimal _tax;
    public decimal Total
    {
        get {
            return _total;
        }
    }

    // Enviamos el porcentaje del impuesto
    public SaleWithTax(decimal tax)
    {
        _tax = tax;
    }

    // Sobreescribimos el método Add para multiplicar la cantidad
    // por el porcentaje del impuesto, e invocamos el método Add
    // de la clase padre con la palabra reservada base
    public override void Add(decimal amount){
        base.Add(amount * _tax);
    }
}
```

Una vez que sobreescribimos un método al momento de crear un objeto de la clase hija el funcionamiento es reemplazado.

```
// Enviamos el porcentaje del impuesto en el constructor
SaleWithTax sale = new SaleWithTax(1.16m);

// El valor de tipo de dato decimal debe tener una m al final
sale.Add(10m);
sale.Add(20m);

// Obtenemos 34.8 como salida en pantalla
Console.WriteLine(sale.Total);
```

Con la sobreescritura tenemos el poder de extender el funcionamiento existente en clases ya hechas, dándonos mayor flexibilidad a la hora de reutilizar código.

Ventajas de la programación orientada a objetos

En programación todo son ventajas y desventajas, y la programación orientada a objetos ha ido generando sugerencias extras a todos sus conceptos.

Entre estas sugerencias se encuentran los principios SOLID y los patrones de diseño, temas que salen del objetivo de este libro, pero vale la pena mencionarlos.

Las ventajas que nos trae el paradigma orientado a objetos son:

- Reutilización de código: podemos reutilizar los objetos así como funcionalidad de estos con la herencia.
- Modularidad: al centrarse en objetos, tenemos el funcionamiento modulado.
- Abstracción: ya que vivimos en un mundo lleno de objetos, nos es fácil abstraer en objetos la funcionalidad de nuestro software.

- Seguridad: gracias al encapsulamiento podemos proteger funcionalidad e información.
- Flexibilidad: gracias al polimorfismo podemos reutilizar funcionamiento y a la vez modificar una parte de este.

Cabe mencionar también desventajas que podrían aparecer al trabajar con este paradigma de programación:

- Complejidad inicial: al inicio debemos organizar en clases el funcionamiento, con lo cual no podemos hacer uso de este rápidamente.
- Mayor consumo de memoria: Al trabajar con objetos desde un inicio, estos heredan funcionamiento el cual quizá nunca se utilice, pero si tiene su lugar en la memoria.

5. Conceptos importantes en la Programación

En la programación tenemos una cantidad enorme de conceptos, que una vez entendidos nos harán la vida más fácil al momento de ir

abordando nuevas tecnologías, ya que estos conceptos se repiten una y otra vez.

En esta sección me centraré en los conceptos que creo que son importantes conocerlos y que bien aprendidos harán que nuestro aprendizaje sea más fácil.

Estructuras de datos

Las estructuras de datos es un tema enorme el cual solo quiero abordar sobre su utilidad, ya que se podría escribir un libro completo sobre este concepto y aun así no abordaremos todo lo que implica el tema.

Las estructuras de datos son formas que sirven para organizar información, como esta se guarda, y como esta se ordena.

En lo que va este libro ya hemos visto algunas estructuras de datos que son los objetos y los array.

Existen más estructuras de datos las cuales tienen reglas y nos servirán dependiendo de situaciones en específico.

Voy a explicar esto con una analogía, cuando tu vas a pagar un supermercado, lo que haces es ir a una fila de personas, esta fila de personas irá disminuyendo según sean atendidas, y en un momento llegará tu turno y saldrás de la fila una vez atendido.

Si quisiéramos organizar información con un comportamiento de fila en supermercado, existe una estructura de datos llamada Queue en inglés, o cola en español, la cual ya tiene estas reglas por sí misma, lo cual nos facilita este comportamiento y organización de información.

En una estructura de datos Queue funciona como un listado de elementos, en el cual el primero en entrar al listado, es el primero en salir, tal cual una fila en el supermercado.

En la programación hay muchas situaciones donde debemos organizar la información en una Queue, por ejemplo, al llegar muchas solicitudes que sobrecarguen los recursos en el Hardware (Memoria RAM, CPU, etc) al ser atendidas de forma paralela, una manera de solventar y no tumbar el sistema es meter la información en una Queue que atienda un elemento a la vez.

Las estructuras de datos son conceptos que se repiten sin importar que es lo que programes, ya sea Backend, Frontend, Videojuegos, Apps móviles, son conceptos los cuales una vez aprendidos no se debe complicar implementarlos en cualquier lenguaje de programación.

Otras estructuras de datos importantes son:

- Stack: conocida en español como pila, el Stack es una estructura parecida a la cola, solo en este caso el primero que entra al listado es el último en salir. Imagina una pila de platos, el último plato que pones en la pila es el primero que tomas, y el primer plato el último ya que está hasta abajo.
- List: la lista es una estructura de datos que tiene su base en el array, solo incorpora métodos que sirven para hacer el acceso, edición, eliminado y agregado de datos más fácil,
- Graph: El grafo es una estructura de datos que tiene muchos tipos, un grafo puede ser dirigido o no dirigido, sirve para modelar cualquier información que especifique relación con otros elementos, por ejemplo si deseas guardar localidades y cómo estas se unen por calles, un grafo te servirá. Otro ejemplo de grafo es una red social, donde los usuarios tienen amigos o siguen personas, esas relaciones se pueden guardar en un grafo.
- Tree: Un árbol es un grafo especial, el cual cuenta con una raíz que tiene 2 hijos y estos hijos a su vez pueden tener otros 2 hijos, parecido a un árbol genealógico. Existen distintos tipos de árboles, uno muy utilizado es un árbol B o en inglés B-Tree, el cual cada que se introduce un elemento se organiza para siempre tener el elemento raíz a su derecha un elemento

mayor y a su izquierda un elemento menor, haciendo que siempre el árbol pueda realizarse una búsqueda óptima, muy utilizado este mecanismo en como las bases de datos indexan los elementos para una búsqueda rápida.

- Sets: los conjuntos o Sets son una estructura de datos que almacena elementos únicos, de esta manera contamos con una colección de información la cual no se va a repetir.

Existen muchas más estructuras de datos, el objetivo es que nazca tu curiosidad por estos temas, que, dicho antes, podría ser un libro por sí mismo.

También aclarar, que es común encontrar ya implementaciones de estas estructuras de datos hechas en todos los lenguajes de programación, algunos como C# ya contienen Stack, List y Queue sin necesidad de instalar nada externo.

Mi recomendación, si eres nuevo en la programación, es que no menosprecies las estructuras de datos, ya que, al dominarlas, son conceptos que utilizarás todos los días como programador y puede ser un diferenciador a la hora de una entrevista técnica para un trabajo como desarrollador.

Ámbito

En la programación el ámbito es el alcance que tiene una variable, clase o método para poder ser utilizado.

En los lenguajes de programación existen reglas para especificar un ámbito, y el entender el alcance de un elemento puede evitar problemas difíciles de rastrear, sobre todo cuando se crean elementos globales que se pueden acceder en cualquier parte de un programa.

A lo largo de mis años como programador, he visto como muchos desarrolladores por no entender este concepto terminan haciendo software el cual es poco legible en cuanto el rastreo de un error, ya que abusan de la creación de variables globales las cuales son

modificadas por muchas secciones del código, ocasionando que sea complejo encontrar que parte es la que está alterando a un valor que ocasiona un mal funcionamiento.

Una variable creada dentro de una función solo existe en dicha función, en un **proyecto de consola de C#** podemos tener funciones sin necesidad de una clase, y toda variable declarada fuera de las funciones es global, lo cual hace que pueda utilizarse en cualquier lugar. Una variable local es una variable que tiene un alcance limitado a donde fue creada, si es creada en una función, su alcance solo es en dicha función.

A continuación, te muestro el alcance de las variables:

```
// Variable num con ámbito global
int num = 1;

void Print1()
{
    // La variable num puede utilizarse, ya que es global
    Console.WriteLine(num);
}

void Print2()
{
    // Se declara una variable local llamada num
    int num = 2;

    // Al existir una variable local num en la función, esta es la
    // que es tomada
    Console.WriteLine(num);
}

Print1();
Print2();
```

```
// Obtendremos como resultado  
// 1  
// 2
```

Las variables tienen un alcance dependiendo en donde son creadas, al ser creadas dentro de una función, su alcance es solo en esa misma función, por lo cual no pueden accederse por fuera.

Al existir dos variables con el mismo nombre, la variable local de la función es la que será tomada en cuenta.

En el caso de las clases tenemos un comportamiento parecido, con la diferencia de que si podemos acceder a una variable que está fuera de una función (método), que en el caso de una clase sería un campo o propiedad.

Vamos a ver un ejemplo de código para explicar el ámbito en clases:

```
// creamos un objeto Some  
Some some = new Some();  
  
// invocamos el método Print()  
some.Print();  
  
// obtendremos la siguiente salida  
// 4  
// 3  
  
class Some  
{  
    // Campo de la clase  
    protected int num = 3;  
  
    public void Print()  
    {  
        // variable local num en el método Print()  
        int num = 4;  
  
        // Al llamar num, se toma la variable local, en caso de
```

```
// que no existiera num local, se tomaría el campo de la  
// clase  
Console.WriteLine(num);  
  
// Para acceder al campo de la clase accedemos con this  
Console.WriteLine(this.num);  
}  
}
```

Null

Cuando creamos una variable en nuestro programa se reserva una dirección de memoria en la computadora, esta dirección de memoria puede tener información o no tenerla, pero cuando haces la variable este espacio es reservado como una caja, para almacenar información.

El concepto de Null es tener una variable sin información, pero reservando el espacio, imagina una caja que existe, una caja con el tamaño adecuado para el tipo con el que se ha definido la variable, pero sin nada dentro, pero eso no quiere decir que la caja no exista.

En Programación Orientada a Objetos al crear una variable sin su inicialización, es decir sin el "**new UnObjeto()**", podemos asignarle el valor **null**, de esta manera estamos definiendo en nuestro código una variable que no tiene información.

```
A a = null;  
  
class A  
{  
    public string Name = "A";  
}
```

Una variable que reserva el espacio para un objeto pero que está en null no puede hacer uso de los métodos, propiedades ni campos del objeto.

Lo siguiente causaría un error:

```
// Asignamos null a la variable del tipo A
A a = null;

// Esta línea causaría un error, ya que la variable es null
Console.WriteLine(a.Name);

class A{
    public string Name = "A";
}
```

El concepto de **null** es algo utilizado por la mayoría de lenguajes de programación, en algunos es llamado **nil** pero sigue la misma lógica.

Null nos da la capacidad de tener variables vacías las cuales más adelante en nuestro código inicializamos por diversas causas, pero recordar que una variable null solo es una caja sin nada dentro.

Parámetros por referencia

El tema de la referencia es uno de los temas más infravalorados que he visto con mis años como programador, y no es un tema complejo, simplemente se ignora.

La referencia en programación es como se pasa una variable a una función, esta variable puede ser pasada por valor o por referencia.

Cuando pasamos datos por valor, la función si hace cambios a la variable internamente, no ocasiona cambios a la variable original.

En cambio, cuando se pasa una variable por referencia, todo cambio hecho en la función o método internamente, también se realiza en la variable original.

Volvamos al ejemplo de la caja, una variable es una caja, que puede tener algo dentro, imagina que cuando pasas por valor, lo que se pasa es el contenido interno, si tenías un numero en tu caja al pasarse por valor a un método, lo que hagas con ese número no

afecta a la variable inicial, no afecta a la caja inicial ya que se pasó solo la información y no la caja, no la referencia.

En el caso de pasar un valor por referencia una variable, con el mismo ejemplo, estaríamos pasando la caja, no se crea una copia, y todo lo que hagas con el contenido de la caja dentro de un método afectará a la caja original, ya que es la misma caja, no una copia.

El no comprender esto puede ser un problema que puede ocasionar errores difíciles de rastrear, sobre todo cuando se utiliza programación concurrente.

Nota: La programación concurrente son programas que pueden ejecutar varias tareas independientes y simultáneas, estas tareas pueden compartir recursos, siendo un recurso variable.

En la Programación Orientada a Objetos, todo objeto al pasarse como parámetro en un método pasa automáticamente por referencia, pasa la caja junto con su contenido.

```
// Creamos un objeto al cual
Beer myBeer = new Beer();
// Le asignamos el valor "Erdinger"
myBeer.Name = "Erdinger";

// Al pasar el objeto, este pasa por referencia, todo cambio en
// el método ChangeName() es realizado en el objeto original.
ChangeName(myBeer, "Corona");

// Al mostrar en pantalla, veremos que se ha cambiado el objeto
// Pantalla muestra: Corona
Console.WriteLine(myBeer.Name);

// Método que solo cambia el valor de Name del objeto Beer
void ChangeName(Beer myBeer, string newNameBeer)
{
    // Todo objeto pasa por referencia, se modifica el original
    myBeer.Name = newNameBeer;
}
```

```
// Clase cerveza con propiedad Name pública  
class Beer  
{  
    // podemos declarar propiedades que no dependan de un campo  
    // de esta manera, agregando al final { get; set; }  
    public string Name { get; set; }  
}
```

Muchas veces por no comprender este concepto de paso por referencia, junto a que todo objeto pasa por referencia, puede ocasionar mucho error al crear software. Entender esto te ayudará a evitar problemas graves y difíciles de rastrear.

Parámetros por valor

En el tema anterior vimos cómo los objetos al pasar por parámetro, estos se pasan por referencia, y ahora te preguntarás ¿Cuándo pasa algo como valor?

En el caso del lenguaje de programación C# existen otras entidades a parte de los objetos las cuales son llamadas structs.

Los structs son tipos que sirven para almacenar valor, y no referencia.

Ya vimos algunos tipos struct que son el tipo int y bool, los cuales, si los pasamos como parámetro en un método, estos pasarían por valor, se crea una copia de su contenido, y todo lo que pase dentro de un método no afecta a la variable original.

```
int num = 8;  
bool isAlive = true;  
  
// Mandamos las variables por valor dada su naturaleza como struct  
Change(num, isAlive);  
  
// No se modifican las variables están fuera del método  
// Se imprime en pantalla:
```



```
// 8
// true
Console.WriteLine(num);
Console.WriteLine(isAlive);

void Change(int num, bool isAlive)
{
    num = 9;
    isAlive = false;
}
```

Inmutabilidad

En la programación, algo inmutable es algo que no puede ser modificado.

Esto sirve en programación en situaciones donde un valor no tiene que cambiar por nada, por ejemplo, un valor constante, siendo una constante quizá una fórmula matemática.

La inmutabilidad es un concepto que puede aplicarse no solo al codificar, también puede aplicarse en distintos entornos como el blockchain que maneja este concepto, o igual lo podemos encontrar en copias de seguridad inmutables.

Este concepto lo podemos encontrar en la Programación Orientada a Objetos y también en otros paradigmas como el paradigma funcional que veremos en la siguiente sección.

Un objeto inmutable es un objeto que una vez creado no puede ser modificado.

En los lenguajes de programación tendremos prácticas y funcionamientos que nos apoyaran para que algo sea inmutable.

Una práctica para crear propiedades inmutables es solo asignarles un método get y no un método set:

```
Beer myBeer = new Beer("Stout");
```

```
// Podemos acceder al valor
string name = myBeer.Name;

// No podemos editar, ya que no existe método set en la propiedad
myBeer.Name = name;

class Beer
{
    // La propiedad solo tiene get, y no tiene set
    public string Name { get; }

    // Se agrega el valor al crear el objeto
    public Beer(string name)
    {
        Name = name;
    }
}
```

El ejemplo anterior muestra una práctica con la cual podemos hacer que una propiedad sea inmutable.

En algunos Lenguajes de Programación tenemos la característica de hacer una propiedad de solo lectura, en el caso de C# cuenta con dicha característica.

```
People hector = new People("Héctor");

// Podemos acceder al valor
string nameHector = hector.Name;

// No podemos editar, ya que es de sólo lectura
hector.Name = "Pedro";

class People
{
    // Con la palabra readonly hacemos la propiedad de solo lectura
    public readonly string Name;

    public People(string name)
```

```
{  
    Name = name;  
}  
}
```

La inmutabilidad también puede aplicarse en los métodos, un método inmutable es aquel que no modifica el contenido del objeto, en su lugar regresa un nuevo objeto con los cambios hechos.

En cambio, un método mutable es aquel que modifica al objeto al que pertenece.

Entender que son los métodos mutables e inmutables es básico para evitar problemas a la hora de programar sistemas que comparten recursos, por ejemplo, programación paralela.

Vamos a ver un ejemplo a continuación, utilizando una clase llamada Sale:

```
class Sale  
{  
    public decimal Total { get; set; }  
    public Sale(decimal total)  
    {  
        Total = total;  
    }  
    // Método inmutable, no altera al objeto en uso, crea una  
    // copia  
    public Sale AddAmountImmutable(decimal amount)  
    {  
        // Creamos un nuevo objeto con el mismo valor en Total  
        Sale newSale = new Sale(this.Total);  
    }  
}
```

```

        // Sumamos en el nuevo objeto
        newSale.Total += amount;

        // regresamos el nuevo objeto
        return newSale;
    }

    // Método mutable, modifica los valores del objeto en uso
    public Sale AddAmountMutable(decimal amount)
    {
        // Se modifica el Total del objeto en uso
        Total += amount;
        // Se regresa el objeto en uso con la palabra this
        return this;
    }
}

```

Vamos a crear un objeto de la clase anterior para ver las diferencias en métodos mutables e inmutables:

```

// Creamos el objeto con valor 10 al inicio
Sale sale = new Sale(10m);

// invocamos el método que no modifica al objeto, crea una copia la
// cual tendrá 11, y el objeto original mantiene 10
Sale immutableSale = sale.AddAmountImmutable(1m);

// Invocamos el método que si modifica al objeto, tendríamos el
// valor 20 en la propiedad Total del objeto original que tenía 10
Sale mutableSale = sale.AddAmountMutable(10m);

// Al invocar los dos objetos tienen valores distintos
// En pantalla tendríamos:
// 11, resultado de los 10 iniciales + 1 en método inmutable
// 20, resultado de 10 + 10, no le afecta el 1 del método
// inmutable
Console.WriteLine(immutableSale.Total);

```

```
Console.WriteLine(mutableSale.Total);
```

La inmutabilidad existe para situaciones donde necesitemos mantener un objeto con sus valores originales, en casos donde el objeto es compartido por muchos procesos, es importante este concepto de inmutabilidad, sobre todo en concurrencia.

Veremos más adelante más sobre inmutabilidad cuando estemos en la sección de programación funcional.

Azúcar sintáctica

El nombre de azúcar sintáctica es un término que puede sonar cómico al inicio, pero tiene su nacimiento en los años 60 gracias a Peter J. Landin uno de los pioneros de la programación funcional.

La azúcar sintáctica es darle características a un lenguaje de programación para escribir más con menos, no es funcionalidad nueva, simplemente una forma de escribir menos código que lleva al mismo resultado.

En C# tenemos azúcar sintáctica en muchas situaciones, por ejemplo al nombrar una variable podemos hacerlo de la siguiente 3 formas:

```
// Creación normal escribiendo el nombre del objeto 2 veces  
A myObject1 = new A();  
  
// Con var solo basta definir el nombre del objeto a la derecha  
var myObject2 = new A();  
  
// Escribir el nombre del objeto al inicio y solo basta con poner  
// new()  
A myObject3 = new();  
  
class A {  
}
```

Las 3 formas de crear un objeto tienen el mismo funcionamiento, solo estamos escribiendo menos código en el ejemplo 2 y 3, evitando escribir el nombre del objeto 2 veces.

Otro ejemplo de azúcar sintáctica en C# es a la hora de escribir métodos que solo tendrán 1 línea de código.

```
class B{  
    // Normalmente escribimos llaves de apertura y cerradura  
    public void Some1()  
    {  
        Console.WriteLine("Algo pasa");  
    }  
  
    // Si solo se tendrá una sola línea de código podemos ahorrar  
    // las llaves {} por => y hacerlo todo en una sola línea  
    public void Some2() => Console.WriteLine("Algo pasa");  
}
```

También podemos encontrar azúcar sintáctica a la hora de concatenar string y variables con un concepto llamado interpolación de cadenas o en inglés interpolation string, podemos hacerlo de la siguiente manera:

```
string name = "Héctor";  
string country = "México";  
  
// Normalmente lo hacemos con el símbolo +  
string message1 = "Hola " + name + " de " + country;  
  
// Podemos iniciar la cadena de texto con $, eso nos permitirá  
// agregar las variables entre {} si necesidad de cerrar comillas  
string message2 = $"Hola {name} de {country}";
```

Existen más características de azúcar sintáctica en C#, veremos algunas más en la siguiente sección de programación funcional.

Cada lenguaje de programación tiene sus propias características, la azúcar sintáctica es parte de la evolución de un lenguaje de programación, con el objetivo de escribir menos código y a su vez no pierda su legibilidad.

6. Programación Funcional

¿Qué es la programación funcional?

En el mundo de la programación existen distintos paradigmas, siendo un paradigma un estilo de programación que tiene sus practicas y reglas.

Ya vimos una sección completa del paradigma de Programación Orientado a Objetos, el cual su base es el objeto y sus reglas como el encapsulamiento para protección, herencia para reutilización.

En la Programación Funcional la base es la función y las reglas son simples, que cada función no debe alterar nada que esté fuera de ella, va muy de la mano de la inmutabilidad.

Este paradigma nace en los años 60s tomando como base el cálculo lambda de Alonzo Church.

Hay lenguajes de programación que se centran en el paradigma funcional, como LISP el primer lenguaje de programación funcional, u otros como Haskell y Clojure que cuentan con características para hacer más fácil este paradigma.

En el mundo laboral los lenguajes anteriormente mencionados no son tan utilizados a comparación de JavaScript, C#, Java, PHP, Python etc. Pero gracias a que la Programación Funcional otorga ventajas en ciertos escenarios como su fácil entendimiento, los lenguajes de programación más utilizados han ido adaptando características de la programación funcional.

Los conceptos que veremos en esta sección pueden ser aplicados en cualquier lenguaje de programación, son conceptos que te servirán en tu vida diaria como programador.

Programación imperativa vs programación declarativa

La programación que hemos visto hasta este momento se basa en ir diciendo “¿Como hacer algo?” mejor conocida como programación imperativa.

Por ejemplo, si deseamos imprimir en pantalla los números pares de un array de números haríamos lo siguiente:

```
// Podemos añadir elementos iniciales a un array utilizando {}  
int[] numbers = new int[10]{  
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
};  
  
// Programación Imperativa  
for (int i = 0; i < numbers.Length; i++)  
{  
    // El operador % obtiene el residuo de una división  
    // 2 % 2 = 0  
    // 2 % 3 = 1  
    if (numbers[i] % 2 == 0){  
        Console.WriteLine(numbers[i]);  
    }  
}
```

En el ejemplo anterior estamos programando de forma imperativa, indicando como llegar al resultado, indicamos por medio del bucle for “**¿Cómo?**” recorreremos el array, le indicamos que a partir del número 0 recorramos hasta la posición menor al tamaño de nuestro array.

Este tipo de programación es más cercana a cómo funciona una máquina, donde indicamos todos los pasos a seguir de manera manual.

La programación funcional es un tipo de programación declarativa la cual se centra en el “¿Qué?” y no en el “¿Cómo?”, entender esto es parte esencial de la programación funcional.

La programación funcional forma parte de los paradigmas declarativos, es un estilo de programación que se combina con programación orientada a objetos, es común que laboralmente vayamos a utilizar los 2 enfoques.

No necesitas entender a fondo lo siguiente por ahora (más adelante explicaremos a detalle los conceptos), pero te mostraré como podemos hacer el mismo resultado del ejemplo anterior utilizando solo programación declarativa, indicando “¿Qué?” es lo que deseamos, más no como hacerlo:

```
// Podemos añadir elementos iniciales a un array utilizando {}  
int[] numbers = new int[10]{  
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
};  
  
// Programación Declarativa  
// Es una sola línea de código, solo no ha alcanzado en la página  
numbers.Where(n => n % 2 == 0).ToList()  
    .ForEach(n => Console.WriteLine(n));
```

El código anterior obtiene el mismo resultado que el código imperativo.

Lo que estamos haciendo es una transformación de la información por medio de funciones que van encadenando su resultado.

A el array de números le estamos indicando con la función Where() que nos dé solo los números que su residuo cuando restamos con el número 2 sea 0, esos son los números pares, posterior convertimos los números pares ya obtenidos a un tipo List de C# con el método ToList(), el tipo List de C# tiene un método que nos sirve llamado

ForEach() el cual hace un recorrido por todos los elementos y a su vez realizar algo por cada elemento individual, y dentro de la función ForEach() hemos definido Console.WriteLine() el cual se ejecutará por cada número par de la colección.

Si notas la diferencia con el paradigma funcional que es parte de la programación declarativa, es que no estamos indicando ¿Cómo? hacer las cosas, estamos indicando ¿Qué? es lo que deseamos.

Con las funciones vamos transformando la información para llegar al objetivo deseado. Leer este tipo de código es muy parecido a como pensamos, con la práctica en programación funcional nos será muy fácil y rápido entender este tipo de código.

En el mundo laboral, vamos a necesitar entender tanto la Programación Orientada a Objetos como la Programación Funcional, las cuales vamos a utilizar en conjunto.

En algunos escenarios será mejor utilizar Programación Orientada a Objetos, y en algunos otros Programación Funcional.

Función de primera clase

Vamos a comenzar a ver el conjunto de características que existen en la programación funcional comenzando con la función de primera clase.

Una función de primera clase es una función que puede guardarse en una variable, al igual estando ya guardada en una variable, puede pasarse como parámetro a otra función y también ser el resultado de una función.

```
void Hi()  
{  
    Console.WriteLine("¡Hola Mundo!");  
}  
  
// Guardamos la función Hi() en la variable hello
```

```
var hello = Hi;  
  
// Para ejecutar la función guardada solo agregamos paréntesis a  
// la variable  
hello();  
// Se muestra en pantalla:  
// ¡Hola Mundo!
```

Al guardar una función en una variable obtenemos mayor flexibilidad a la hora de programar, teniendo el poder de guardar funcionamiento en una variable.

Función orden superior

Una función de orden superior es una función que obtiene funciones como entrada de datos, es decir funciones guardadas en variables y pasadas en los parámetros, o también una función de orden superior es una función que retorne otra función.

Las funciones de orden superior van de la mano de las funciones de primera clase, ya que dependen del poder guardar funciones en variables.

Una función guardada en una variable puede ser enviada como parámetro y ejecutada dentro de una función de orden superior en su contenido interno en un momento en específico.

En C# las funciones deben tener un tipo, y para ello haremos uso de algo llamado delegado, el delegado lo veremos más a fondo adelante en esta misma sección, pero por ahora basta con que entiendas que sirven para otorgarle un tipo a una función, de esta manera las funciones de orden superior protegen el tipo de función que reciben o el tipo de función que regresan.

```
// Definimos una función que regresa un string  
// estamos utilizando azúcar sintáctica ya que solo es una línea  
string Message() => "Un mensaje externo";
```

```
// DoSomething es una función de orden superior ya que recibe
// una función como parámetro
// Al utilizar Func<string> estamos especificando que la función
// que recibe debe si por si regresar un string
void DoSomething(Func<string> fn){
    Console.WriteLine("Algo se hace antes");

    // Ya que definimos que la función enviada regresa un string
    // podemos imprimir el mensaje que regresa la variable fn()
    Console.WriteLine(fn());
    Console.WriteLine("Algo se hace después");
}

// Ejecutamos la función DoSomething enviándole como parámetro
// la función Message()
// Podemos enviar una función existente sin los parentesis, funciona
// igual que si la guardamos en una variable
DoSomething(Message);
```

En pantalla veremos el siguiente resultado:

```
Algo se hace antes
Un mensaje externo
Algo se hace después
```

En el ejemplo anterior tenemos una función **DoSomething()** que tiene un comportamiento inicial y otro final, entre estos dos comportamientos tiene la flexibilidad de ejecutar algo que desconoce pero sabe que regresará un valor string.

Las funciones de orden superior junto con las funciones de primera clase nos permiten codificar de manera más flexible, brindándonos el poder ejecutar algo ya existente de distintas maneras, ya que una función puede realizar una tarea y extender su funcionamiento permitiendo funciones de entrada como parámetro.

Función pura

Una de las reglas más importantes en la programación funcional es el concepto de función pura.

Una función pura es

- Una función que siempre produce el mismo resultado dados siempre los mismos datos de entrada.
- No tiene efectos o cambios con elementos externos a ella.

La ventaja que tenemos al crear funciones puras es que sabemos que siempre nos va a retornar el mismo resultado con la misma entrada y también estamos seguros de que no altera elementos externos, aquí va muy de la mano de la inmutabilidad y del paso de referencia que vimos anteriormente.

También al tener funciones puras podemos reemplazar una función por otra sabiendo que no va a alterar lugares inadecuados de nuestro programa, a esto se le llama **Transparencia referencial**. Como no tiene efectos con elementos externos y siempre regresa lo mismo dado los mismos datos de entrada, no debe pasar nada raro en nuestro código.

Una parte importante en la programación es la refactorización de código, la cual se realiza para darle mayor calidad al código, modificando el software para hacerlo más legible y escalable sin alterar el funcionamiento. Cuando se tienen funciones puras, puede ayudar bastante a la hora de mejorar el código.

Para comprender una función pura, primero vamos a ver que no es una función pura:

```
// Función impura AddSeconds()
// Función que aumenta los segundos de la hora actual
DateTime AddSeconds(int seconds)
{
    // 1. DateTime.Now nos regresa un valor del struct DateTime
    // con la fecha y hora actual de la ejecución
```

```

    // 2. AddSeconds() es un método de DateTime que aumenta
segundos
    // a un valor DateTime
    DateTime date = DateTime.Now.AddSeconds(seconds);
    return date;
}

// 1. Al ejecutar 2 veces la función vemos distintos resultados
// dado el mismo valor de entrada, por lo cual AddSeconds no es
// función pura.
// 2. TimeOfDay es una propiedad existente en el struct DateTime.
// 3. AddSeconds() nos regresa un DateTime por lo cual podemos
usar
// enseguida las propiedades de DateTime.

Console.WriteLine(AddSeconds(1).TimeOfDay);
Console.WriteLine(AddSeconds(1).TimeOfDay);

// En pantalla veríamos valores distintos:
// 16:43:24.8542065
// 16:43:24.8867146

```

La función anterior es una función impura ya que no cumple con una de las reglas de funciones puras que es regresar siempre el mismo resultado bajo la misma entrada de información.

Esto pasa ya que internamente calcula la fecha actual, con lo cual el valor interno cambia, el tiempo avanza y cada que ejecutemos dicha función obtendremos valor distinto a pesar de mandar el mismo valor como parámetro de entrada.

Para hacer la función pura podemos hacer lo siguiente:

```

// Función pura
// La fecha la obtenemos fuera de la función
var now = DateTime.Now;

// La función DateAddSeconds() es función pura, obtiene el valor

```

// de la fecha como parámetro, así no se calcula internamente evitando valores distintos en cada ejecución.

```
DateTime DateAddSeconds(DateTime date, int seconds)  
{  
    date.AddSeconds(seconds);  
    return date;  
}
```

```
Console.WriteLine(DateAddSeconds(now, 1).TimeOfDay);  
Console.WriteLine(DateAddSeconds(now, 1).TimeOfDay);
```

// El resultado en pantalla mostraría el mismo valor
// 16:52:46.2829192
// 16:52:46.2829192

La función anterior, es una función pura ya que el cálculo de la fecha actual se hace de manera externa, así se evita estar calculando internamente valores que cambian con el tiempo, de esta manera la función actúa siempre igual para los mismos valores de entrada.

Ahora veamos otro ejemplo de una función impura:

```
// Clase Beer para el ejemplo  
class Beer  
{  
    public string Name { get; set; }  
}
```

El código de ejecución:

```
// Hay otra manera de dar valores a los objetos: escribiendo entre  
// llaves después de la creación del objeto {Pro1=value,  
Pro2=value}  
var beer = new Beer(){  
    Name = "Colimita"  
};
```



```

// Función impura
// ToUpperName() es una función impura ya que al recibir un objeto
// como parámetro, este va por referencia, todo cambio interno se
// realiza al objeto original
Beer ToUpperName(Beer beer){
    // Se modifica el objeto original
    beer.Name = beer.Name.ToUpper();
    return beer;
}

// Al mostrar el resultado del objeto retornado por la función
// y el objeto original veremos el mismo valor en los dos.
Console.WriteLine(ToUpperName(beer).Name);
Console.WriteLine(beer.Name);

// En pantalla veremos:
// COLIMITA
// COLIMITA

```

La función anterior es una función impura ya que modifica un objeto externo a ella, rompiendo la segunda regla de funciones puras: no alterar elementos externos.

Al alterar elementos externos podemos alterar secciones del software que serían difíciles de rastrear, lo cual ocasionará tiempo extra a la hora de buscar un error.

Para hacer que la función sea pura haríamos lo siguiente:

```

var beer = new Beer(){
    Name = "Colimita"
};

// Función pura
// BeerToUpperName() es una función pura, ya que crea un nuevo

```

```
// objeto internamente, el objeto interno es el que se modifica  
// y regresa, sin alterar el objeto original.  
Beer BeerToUpperName(Beer beer)  
{  
    // Se crea un nuevo objeto con el valor de Name  
    // del objeto recibido.  
    var newBeer = new Beer(){  
        Name = beer.Name.ToUpper()  
    };  
    return newBeer;  
}  
  
// Los valores serán distintos  
Console.WriteLine(BeerToUpperName(beer).Name);  
Console.WriteLine(beer.Name);  
  
// En pantalla veremos:  
// COLIMITA  
// Colimita
```

La función anterior se ha modificado para crear un nuevo objeto internamente al cual se realizará la modificación, de esta manera cumplimos con la regla de no tener efectos colaterales, de no modificar elementos externos a la función.

Las funciones puras nos permiten poder asegurarnos de no tener efectos impredecibles, cabe recalcar que no siempre es posible crear funciones puras, hay escenarios donde las funciones necesitan de elementos externos, es por eso la importancia de comprender tanto la programación orientada a objetos como programación funcional, y trabajar en conjunto, así es esto de la programación, distintas herramientas que son más adecuadas dependiendo el escenario que nos enfrentemos.

Funciones anónimas (Expresiones lambda)

Una función anónima es aquella que no cuenta con un nombre y puede ser guardada en una variable.

Cuando utilizamos funciones de orden superior, funciones que reciben de entrada otras funciones o retornan una función como resultado, el no estar obligados a definir una función la cual será pasada como parámetro nos permite mayor velocidad al codificar.

En C# Existe algo llamado “**Expresiones Lambda**” las cuales son azúcar sintáctica para definir funciones anónimas.

```
// En una variable llamada some guardamos una función anónima  
var some = () => Console.WriteLine("Hola mundo");  
  
// Para ejecutar la función basta con poner paréntesis a la  
// variable  
some();
```

En el ejemplo anterior hemos creado una variable llamada some la cual guarda una función que muestra un mensaje, esta función no tiene nombre, es una función anónima guardada en la variable some.

Para definir una función anónima que reciba parámetros podemos hacerlo de la siguiente manera:

```
// Entre los paréntesis definimos los parámetros, no hay necesidad  
// de especificar el return ya que es una función de una sola línea  
var add = (int a, int b) => a + b;  
// Se ejecuta invocando la variable como si fuera una función  
int res = add(1,2);  
Console.WriteLine(res);
```

Las funciones anónimas con más de una línea de código deben tener llaves como una función normal y si regresan un valor deben utilizar return:

```
var mul = (int a, int b) => {  
    int res = a * b;  
    return res;  
};
```

La ventaja de utilizar expresiones lambda para crear funciones anónimas es no tener la necesidad de definir funciones y poder mandar la función directamente a una función de orden superior que recibe funciones como entrada en sus parámetros.

Nota: En JavaScript las funciones anónimas pueden escribirse con funciones flecha o arrow function en inglés, las funciones flecha tiene la misma sintaxis que las expresiones lambda de C#.

Tipado de funciones

En C# siendo un lenguaje de programación tipado, esto significa, que se debe definir los tipos de las variables, las funciones no quedan fuera de estas reglas.

Para definir el tipo de una función en C# tenemos algo llamado los delegados.

Existen 3 delegados que nos ayudarán a definir el tipo de nuestra función, que son:

- Func<>: para funciones que regresan un valor.
- Action<>: para funciones que no regresan ningún valor.
- Predicate<>: para funciones que regresan un tipo lógico, un tipo bool, verdadero o falso.

Ya hemos visto un poco del delegado Func, el cual regresa un valor el cual lo especificamos:

```
Func<string> message = () => "Es un mensaje";
```

Con Func<> definimos el tipo de nuestra variable, al utilizar Func especificamos que la función va a regresar un valor, eso lo especificamos entre <tipo de valor> y la función que se guarde en la variable debe cumplir con lo especificado.

El tener tipado en las funciones nos permite seguridad al programar, ya que con el lenguaje de programación se asegura que se le proporcione algo válido, y no una función que no cumpla con lo especificado.

Func también puede especificar un tipo de función que reciba parámetros:

```
Func<int, string> hi = (int number) => $"El número es: {number}";
```

El orden de los parámetros y el tipo de dato que regresa una función especificada con Func es el siguiente:

```
Func<parametro1, parametro2, parametro3, valor que regresa>
```

De esta manera podemos especificar una función con múltiples parámetros y un valor de retorno, con Func solo es obligatorio especificar el valor de retorno, y los parámetros son opcionales.

El último valor especificado en Func es el que será tomado como valor que regresa la función.

Anteriormente vimos ejemplos de esto mismo, pero utilizando la palabra reservada **var**, **var** define el tipo de dato de la variable con el contenido que se especifica a la derecha, C# analiza tu función e identifica el tipo, por lo cual es opción utilizar var.

```
var hi2 = (int number) => $"El número es: {number}";
```

Para definir una función de orden superior, una función que recibe parámetros es necesario especificar el tipo que es el que será recibido:

```
// Se define un parámetro de una función que regresa un string
var messages = (Func<string> fn) => {
    Console.WriteLine("Mensaje inicial");
    Console.WriteLine(fn());
    Console.WriteLine("Mensaje final");
};
// Ejecutamos con función anónima escrita con expresión lambda
messages(()=>"Mensaje central");

// En pantalla se muestra:
// Mensaje inicial
// mensaje central
// Mensaje final
```

En el ejemplo anterior utilizamos una expresión lambda para enviar la función que cumple con el tipo esperado, y así evitar crear una función con nombre.

Existen otros 2 delegados útiles, en el caso de Action<>, funciona igual que Func, con la diferencia de que se definen funciones que no regresan valor:

```
// Definimos una función que no regresa nada, pero recibe un string
Action<string> print = (string msg) => Console.WriteLine(msg);
print("Un mensaje");
```

El otro delegado es Predicate<> el cual sirve para definir funciones que solo reciban un parámetro y regresen un tipo bool:

```
// Definimos una función que verifica si el número dado es par
Predicate<int> isPair = (int number) => number % 2 == 0;
if (isPair(2)){
```

```
Console.WriteLine("Es par");  
}
```

En el caso de Predicate<>, solo puede recibir un parámetro, pero siempre regresará un valor **bool**.

El tener tipado de funciones nos ayudará a crear código más entendible y evitar problemas al mandar funciones que ocasionen comportamientos inesperados.

Recursividad

La recursividad es una técnica en la cual una función se llama así misma.

Esta técnica debe tener una condición para volverse a llamar, y cuando esa condición no sea cumplida, la ejecución de la función termina.

Podemos recorrer un array con esta técnica de la siguiente manera:

```
int[] numbers = new int[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
// Si no se manda valor a index, toma 0 por defecto  
void print(int[] numbers, int index = 0){  
  
    // Comparamos que el índice no sea mayor a la longitud  
    // del array  
    if (index >= numbers.Length){  
        // Terminará la ejecución cuando index sea igual o mayor  
        // a la longitud del array  
        return;  
    }  
  
    // Se imprime el valor de la posición de index  
    Console.WriteLine(numbers[index]);  
  
    // Llamamos Recursivamente la misma función aumentando el  
índice
```

```
    print(numbers, index + 1);  
}  
  
print(numbers);  
// Tendremos en pantalla lo siguiente:  
// 1 2 3 4 5 6 7 8 9 10
```

La recursividad debe manejarse con cuidado, ya que el codificar situaciones donde se tienda excesivas repeticiones puede ocasionar problemas en tus sistemas.

Closures (Clausuras)

El concepto de closure o clausura es un concepto donde una función tiene variables las cuales puede mantener sus valores, también las clausuras regresan una función como resultado.

Esto es útil para compartir información en la ejecución sin necesidad de tener que crear variables externas, o variables que se comparten con otras funciones.

Una clausura tiene la siguiente estructura:

```
// Se define una función que regresara otra función como resultado  
// la función de resultado es una función que recibe un string  
Action<string> print(){  
    // Creamos una variable entera  
    int counter = 0;  
    // Regresamos como resultado una función que recibe un string  
    return (string message) =>{  
        // En cada ejecución de la función retornada se aumenta  
        // counter  
        counter++;  
        // Imprimimos en pantalla el valor de counter y el string  
        Console.WriteLine($"Ejecución {counter} {message}");  
    };  
}
```



```
// 1. Creamos 2 variables con el closure print, cada variable
// mantendrá su valor guardado en counter de forma independiente
// 2. Show regresa una función lista para ejecutarse
var show1 = print();
var show2 = print();
show1("Un mensaje");
show1("Otro mensaje");
show2("Un mensaje con show2");
```

En pantalla tendríamos:

```
// En la pantalla nos muestra:
// Ejecución 1 Un mensaje
// Ejecución 2 Otro mensaje
// Ejecución 1 Un mensaje con show2
```

El objetivo de las clausuras es poder crear funciones las cuales mantengan un estado (valor) y este pueda ser modificado en cada ejecución.

Currificación

La currificación es una técnica con la cual podemos convertir una función que recibe múltiples parámetros en una secuencia de funciones.

El nombre de currificación o curry es tomado del apellido de Haskell Curry, uno de los pioneros de la programación funcional.

La currificación sirve para tener funciones con valores guardados y poder reutilizarlas con el valor inicial, por ejemplo, en fórmulas matemáticas.

Para lograr esto vamos a hacer uso de funciones de orden superior, las cuales regresan funciones como resultado:

```
// La función Mul regresa una función que recibe un parámetro
// float y regresa otra función con entrada y respuesta float.
```

```

Func<float, Func<float, float>> Mul(){
    // Se cumple con el tipo esperado:
    // Func<float, Func<float, float>>
    // Se recibe un float como parámetro, y se regresa otra
    // función Func<float, float>
    return (float a) =>{
        // Se regresa una función que cumple con:
        // Func<float, float>
        // Recibe un float como entrada y regresa un float
        return (float b) => a * b;
    };
}

```

A continuación, te muestro la ejecución de la curificación utilizando la función anterior:

```

// En una variable guardamos el resultado de Mul que es:
// Func<float, Func<float, float>>
var m = Mul();

// Teniendo en m una función ejecutamos enviando un valor float
// para obtener el siguiente resultado que es:
// Func<float, float>
// Los valores float deben tener una f al final del número
var mx10 = m(10f);

// Teniendo en mx10 una función del tipo Func<float, float>, que
// tiene un valor de entrada float y regresa un float, le enviamos
// un valor 9
var res1 = mx10(9f);

Console.WriteLine(res1);
// En pantalla tendremos un 90.

```

En el ejemplo anterior guardamos en **m** la funcionalidad que regresa la función **Mul()**, que al ejecutar nos regresa otra funcionalidad en

mx10 pero con un valor ya establecido que es el número 10, la funcionalidad guardada en **mx10** al ejecutarse ya tiene el valor 10 en la variable **a**, por lo cual al enviar un valor este se incrustara en **b**, y podrá hacerse la multiplicación.

Con la currificación podríamos reutilizar el funcionamiento con valores ya establecidos para realizar alguna acción, por ejemplo, podríamos seguir utilizando la variable **mx10** con otros números:

```
Console.WriteLine(mx10(1f));  
Console.WriteLine(mx10(2f));  
Console.WriteLine(mx10(3f));  
// En pantalla tendremos un 10, 20, 30.
```

La función **Mul()** anterior podemos resumirlo con expresiones lambda de la siguiente forma:

```
Func<float, Func<float, float>> mul = a => b => a * b;
```

Tendríamos el mismo resultado, escribiendo menos código, y se usaría de la misma manera:

```
// Guardamos en circle Perimeter la funcionalidad de mul  
Func<float, Func<float, float>> circlePerimeter = (d) =>  
mul(d);  
  
// Guardamos en fórmula con el valor inicial de 3.1416 para poder  
// utilizarlo en sacar el perimetro de distintos círculos  
var formula = circlePerimeter(3.1416f);  
  
// Guardamos en res2 el perímetro de un círculo con diámetro 10  
var res2 = formula(10f);  
// Guardamos en res3 el perímetro de un círculo con diámetro 20  
var res3 = formula(20f);
```

El objetivo de la currificación es hacer funciones que mantengan información y puedan ser reutilizadas sin necesidad de brindarles

nuevamente el valor inicial.

Ventajas de la programación funcional

Hemos visto algunas características de la programación funcional que puede implementarse en C#, e igual la mayoría de lo antes visto puede ser implementado en los lenguajes de programación no funcionales más utilizados, por ejemplo en Java, JavaScript, PHP, TypeScript o Python, con ciertas diferencias, pero en concepto es lo mismo.

Habrán situaciones donde nos vendrá mejor programar con paradigma orientado a objetos y en otras con paradigma funcional, al final, los dos tipos de programación son necesarios y pueden convivir mutuamente.

Las ventajas de la programación funcional son las siguientes:

- Rastreabilidad de errores: al tener funciones puras podemos ir directo a la ejecución que ocasiona un error.
- Paralelización: al trabajar con sistemas paralelos se presta la programación funcional para evitar cambios en recursos compartidos, por la regla de función pura de no modificar elementos externos a ella.
- Legibilidad: Se cuenta con funciones que realizan una tarea, con lo cual podemos entender rápidamente qué es lo que hace una función.

Al igual existen desventajas de la programación funcional que son:

- Curva de aprendizaje: se tienen menos recursos de programación funcional que de programación orientada a objetos.
- Falta de aceptación: algunos lenguajes de programación están adoptando aún características de programación funcional, podría ser que algún concepto no se aplique aún.
- Problemas al trabajar en equipo: este tipo de prácticas siguen siendo desconocidas para mucho programador, podría

complicarse a un miembro de tu equipo si no entiende lo fundamental de la programación funcional.

El objetivo de esta sección es brindarte la información necesaria de programación funcional para facilitarte tu día laboral como programador, espero haber cumplido con el objetivo.

7. Programación Orientada a Objetos Avanzada

Ya vimos la base de la Programación Orientada a Objetos en una sección anterior, el objetivo de esta sección es centrarnos en conceptos que van un poco más allá de lo básico y que dados desde el inicio podrían ocasionar confusión.

Polimorfismo

El polimorfismo es la capacidad que tiene un objeto para tomar distinta forma, y tener un funcionamiento dependiendo del contexto donde se utilice.

Un ejemplo del mundo real es cuando tu vas a un funeral tienes un comportamiento distinto a cuando vas a un bar por una buena cerveza.

Los objetos tienen la capacidad de comportarse dependiendo del contexto en el que están.

Existen varios tipos de polimorfismos y en sí, encontrarás que dependiendo el autor de algún libro definirá con un nombre o una clasificación distinta a la de otro autor. Pero nos centraremos en el significado de polimorfismo que es que un objeto tenga comportamiento dependiendo su contexto.

Ya vimos algo de esto, lo vimos en la explicación de sobrecarga de métodos. La sobrecarga permite crear métodos en un objeto con el mismo nombre los cuales difieren en los parámetros de entrada otorgados. Dependiendo de los valores que envíes al método, se ejecuta el método adecuado.

El polimorfismo es eso, un comportamiento dependiendo la situación, pero hay otras formas de realizar polimorfismo, por ejemplo, con polimorfismo por herencia.

Polimorfismo por herencia

La herencia nos permite realizar polimorfismo, hacer que un objeto se comporte dependiendo su clase con la que fue creado.

Cuando tenemos una clase padre la cual tiene clases hijas que han heredado de la primera, podemos obtener el comportamiento creado en las clases derivadas.

Al tener una variable creada con la clase padre, podemos asignarle en su creación una clase hija, y este objeto se comportará como fue escrito el código en la clase hija.

Teniendo las siguientes 3 clases, de las cuales la clase B y C heredan de A, y sobrescribe el método `GetClassName()`.

```
class A{  
    public virtual string GetClassName() => "Soy A";  
}  
  
class B : A{  
    public override string GetClassName() => "Soy B";  
}  
  
class C : A{  
    public override string GetClassName() => "Soy C";  
}
```

Podemos hacer lo siguiente, y nuestro objeto cambiará su comportamiento:

```
A myObject = new A();  
Console.WriteLine(myObject.GetClassName());  
myObject = new B();  
Console.WriteLine(myObject.GetClassName());  
myObject = new C();  
Console.WriteLine(myObject.GetClassName());
```

```
// En pantalla tendremos lo siguiente:  
// Soy A  
// Soy B  
// Soy C
```

Con el polimorfismo por herencia podemos cambiar el comportamiento de nuestro objeto en tiempo de ejecución, dándonos mayor flexibilidad a la hora de utilizar programación orientada a objetos.

Interfaces

Una **interface** en Programación Orientada a Objetos es una entidad que nos servirá como contrato, tiene un conjunto de elementos que deben existir en las clases que la implementen.

Una **interface** especifica propiedades y métodos que deben definirse en las clases derivadas, una interface no contiene funcionamiento regularmente (En C# ya se puede incluir funcionamiento inicial).

Lo siguiente es una interface:

```
interface IProduct  
{  
    public decimal Price { get; set; }  
    public string Name { get; set; }  
  
    public string GetInfo();  
}
```

Una interface resulta muy parecida a una clase, solo que su objetivo no es incluir comportamiento, su objetivo es categorizar clases y permitir organizar código en sistemas, esto para poder tener organización de abstracción (más adelante veremos que es abstracción).

La interface anterior **IProduct** nos indica que todo lo que sea un producto está obligado a tener una propiedad Price y Name y un método GetInfo(), obligando a que se respete estos elementos en toda clase que implemente IProduct.

Para implementar una interface se hace de la siguiente manera:

```
// Para implementar se realiza al igual que la herencia con :  
class Beer : IProduct  
{  
    public decimal Price { get; set; }  
    public string Name { get; set; }  
    public string GetInfo() =>  
         $"La información es {Name} ${Price}";  
}
```

Si no se definen los elementos de la interface en la clase que la implementa nos marcará un error antes de ejecutar el programa, de esta manera se protege que el código cumpla con las reglas de la interface, que existan las propiedades y métodos especificados.

Una clase puede implementar varias interfaces, por lo cual podemos categorizarla en distintos funcionamientos.

Una cerveza aparte de ser un producto también es algo que puede beberse, vamos a escribir una interface llamada IDrinkable que especifique eso:

```
interface IDrinkable  
{  
    public decimal Quantity { get; set; }  
    public bool HasAlchol { get; set; }  
}
```

Podemos implementar tanto de IDrinkable como de IProducto, siempre y cuando cumplamos con las reglas que nos imponen las dos interfaces.

Nuestra clase Beer quedaría de la siguiente forma:

```
// Indicamos con coma cuando implementaremos de varias interfaces
class Beer : IProduct, IDrinkable
{
    public decimal Price { get; set; }
    public string Name { get; set; }
    public decimal Quantity { get; set; }
    public bool HasAlchol { get; set; }
    public string GetInfo()
    {
        return $"La información es {Name} ${Price} " +
            "y es un producto bebible con {Quantity} ml";
    }
}
```

Gracias a las interfaces podemos ir creando categorización de nuestro código, de esta manera podemos organizar qué funcionalidades tendrá nuestro sistema, e ir seccionando en interfaces que pueden ser utilizadas de la siguiente manera:

```
var beer = new Beer(){
    Name = "Erdinger",
    Price = 8.5m,
    Quantity = 500,
    HasAlchol = true
};
```

```
void Show(IProduct product){  
    Console.WriteLine(product.GetInfo());  
}  
  
Show(beer);
```

Con las interfaces podemos crear funcionamiento que esperen una implementación, sin que nos importe qué es lo que haga el método internamente, al saber que esperamos en la función Show() un objeto creado con una clase que implementa la interface IProduct, sabemos que el objeto enviado va a estar obligado a cumplir con las propiedades o métodos de la interface definida.

Podemos tener clases que sean producto, pero no sean algo bebible. Por ejemplo, una consola de videojuegos:

```
class VideoGameConsole : IProduct  
{  
    public decimal Price { get; set; }  
    public string Name { get; set; }  
    public string GetInfo()  
    {  
        return $"La información de la consola es {Name}  
        ${Price}";  
    }  
}
```

Y los objetos que se creen con esta clase cumplirán con IProduct por lo cual podrán utilizar la función Show() la cual espera algo que implemente de IProduct:

```
var nintendoSwitch = new VideoGameConsole()  
{  
    Name = "Nintendo Switch",  
    Price = 2000  
};
```

```
void Show(IProduct product)
{
    Console.WriteLine(product.GetInfo());
}

Show(nintendoSwitch);
```

Podemos mandar un objeto sin relación alguna con la clase Beer, a una función que espera un objeto que implemente IProduct.

Las interfaces son la base de los patrones de diseño, los patrones de diseño son técnicas que sirven para resolver problemáticas ya probadas, brindándonos organización para crear, estructurar y establecer comportamiento de clases.

En este libro no hablaremos de patrones de diseño, pero si entendiste las interfaces, será algo que ya no te va a parecer tan complicado como suena.

Generics

Los genéricos en programación sirven para que un método pueda funcionar con distintos tipos de datos, esto nos permite reutilizar funcionamiento, evitando tener que escribir un método por cada tipo de dato con el que necesitemos trabajar.

Los genéricos pueden trabajar a nivel de función, teniendo dos arrays de distintos tipos:

```
int[] numbers = new int[5]
{
    1, 2, 3, 4, 5
};

string[] names = new string[5]{
    "Héctor", "Pedro", "Ana", "Paula", "Juan"
};
```

Si necesitamos recorrer y mostrar su información, podemos hacerlo solo con una función, la cual utilice genéricos de la siguiente manera:

```
// <T> sirve para definir que se enviará un tipo, y ese tipo tiene  
// como alias T, donde representes T será puesto el tipo enviado,  
// en este caso se especifica el parámetro de entrada data.  
  
void Show<T>(T[] data){  
    for (int i=0; i<data.Length; i++){  
        Console.WriteLine(data[i]);  
    }  
}
```

La función anterior nos sirve para cualquier array, sin importar su tipo:

```
// Al especificar <tipo> la función se puede reutilizar  
Show<int>(numbers);  
Show<string>(names);  
  
// En las nuevas versiones de C#, ya no es necesario especificar  
// el tipo al ejecutar la función, C# lo detecta en automático,  
// solo en la ejecución, no en la creación del funcionamiento.  
Show(numbers);  
Show(names);
```

También los genéricos pueden funcionar a nivel clase:

```
// Especificamos el tipo entre <>  
class MyList<T>{  
    // Definimos que el tipo recibido será el de este array  
    private T[] data;  
    // index servirá para llevar la posición del último elemento  
    private int index = 0;  
  
    // Recibimos en el constructor la longitud del array y lo
```

```

// creamos con el tipo recibido y representado con T
public MyList(int length) => data = new T[length];

// Recibimos un elemento del tipo genérico
public void Add(T item){
    data[index++] = item;
}
// Muestra la información del array
public void Show(){
    for (int i = 0; i < data.Length; i++){
        Console.WriteLine(data[i]);
    }
}
}

```

La clase anterior nos sirve para llevar la logística de un array, sin importar su tipo:

```

// Definimos el tipo entre <>
var countries = new MyList<string>(5);
countries.Add("México");
countries.Add("Argentina");
countries.Add("Guatemala");
countries.Show();

```

Al utilizar genéricos podemos reutilizar código, y también organizar nuestra programación de manera más abstracta, identificando funcionamientos que son parecidos y que solo difieren en el tipo de dato.

Foreach

Foreach es una estructura de control del tipo bucle, al igual que For y While que vimos en secciones pasadas.

He querido mostrar esta **estructura de control de bucle** hasta este punto ya que nos servirá para reafirmar el conocimiento ya

obtenido en los temas anteriores.

Foreach sirve para recorrer un objeto que implemente una interface iterable. Ya vimos anteriormente que una interface sirve para definir reglas que deben cumplirse, en cuanto a propiedades y métodos.

En C# existe una interface llamada IEnumerable, una interface que esta en la base del lenguaje mismo y tiene el siguiente código:

```
// Este es código de la base de C#  
public interface IEnumerable  
{  
    IEnumerator GetEnumerator();  
}
```

Toda clase que implemente la interface IEnumerable está obligada a implementar el método GetEnumerator(), esto va de la mano a la sentencia de control de bucle Foreach.

La sentencia Foreach en C# funciona con los objetos que sean de clases que implementen IEnumerable, por ejemplo, existe una clase llamada List la cual trabaja con colecciones de datos, e igual, funciona con genéricos:

```
var numbers = new List<int>(){ 1, 2, 3 };
```

El tipo List de C# siendo una clase que implementa IEnumerable puede utilizarse con la sentencia Foreach para recorrer su información de la siguiente manera:

```
// number es la variable que irá tomando el valor de cada elemento  
// de la lista en cada iteración  
foreach (var n in numbers)  
{  
    Console.WriteLine(n);  
}
```

La sentencia Foreach es una forma muy legible para recorrer colecciones, y es muy utilizada en los lenguajes de programación, evitando la creación de la logística del seguimiento de las iteraciones, como en el caso de la sentencia For donde debíamos llevar el control de una variable hasta el límite de un array.

Foreach no es un reemplazo de las sentencias While o For. Hay situaciones donde es mejor utilizar For o While que Foreach, al final, son herramientas que serán más adecuadas dependiendo el escenario que tengamos que resolver.

Abstracción

El concepto de abstracción en programación es analizar cómo podemos organizar en código la solución de un problema a resolver.

Como la programación orientada a objetos tiene su base en los objetos, la abstracción es esa parte donde identificamos que clases vamos a escribir, las propiedades que tendrán, sus métodos y cómo se van a relacionar en cuanto a herencia, implementaciones y mensajes.

La abstracción es una habilidad que iremos mejorando con la experiencia, encontrando técnicas que nos permitan hacer código más fácil de dar mantenimiento y sobre todo, código que sea legible.

Gracias a la abstracción podemos reutilizar código, hacer código entendible, modularizar código y trabajar con otros programadores en la construcción de software.

Técnicas como los patrones de diseño, y las arquitecturas nos apoyan para abstraer problemáticas ya resueltas por otros programadores.

La abstracción suele ser esa parte que se va adquiriendo entre más codificamos, al final podemos codificar distintas formas de solucionar un problema, pero siempre mejoraremos en hacer código más adecuado para el tipo de proyecto que realizaremos.

Hasta este punto hemos visto los 4 pilares de la programación orientada a objetos:

- Herencia: herencia de clases, implementación de interfaces.
- Encapsulamiento: protección de métodos y propiedades en los objetos.
- Polimorfismo: capacidad que tiene un objeto para actuar de distintas formas dependiendo el contexto.
- Abstracción: identificación de características que serán esenciales en nuestras clases para resolver un problema con código.

Regularmente estos 4 conceptos son mencionados siempre al inicio, pero yo he querido hacerlo al revés, de esta manera creo que puede resultar más entendible, ya que, hasta este punto, ya vimos todos los puntos con ejemplos de código, simplemente les estamos poniendo nombre.

Clase Abstracta

Una clase abstracta es una clase que no puede usarse directamente para crear un objeto.

¿Entonces para qué sirve esto? La clase abstracta sirve de plantilla de otras clases, estas clases pueden tener campos, propiedades y métodos con funcionalidad definida, los cuales pueden ser utilizados por los objetos de las clases que hereden de la clase abstracta.

Una clase abstracta se crea igual que una clase normal, solo agregamos la palabra `abstract`:

```
// La clase abstracta lleva la palabra abstract  
abstract class Liquid  
{  
    public string Name { get; set; }  
    public string Quantity { get; set; }  
}
```

```
}
```

Si intentas crear un objeto con la clase anterior, el código te marcará un error, ya que las clases abstractas solo sirven para abstraer funcionamiento.

El concepto de clase abstracta está relacionado a la abstracción en la programación, el poder identificar partes que serán comunes en un sistema a la hora de diseñarlo, y colocar esas partes en una clase abstracta para no repetir ese código, apoya a que hagamos sistemas más escalables.

Si categorizamos elementos más complejos como cervezas, jugos, licuados que son líquidos, todos estos comparten rasgos en común y características distintas, siendo lo común un nombre y una cantidad.

La abstracción es eso, categorizar las cosas que pueden agruparse para no tener que repetir código.

Para poder utilizar una clase abstracta, debemos heredar de ella:

```
// Se hereda igual que una clase común  
class Beer : Liquid  
{  
    // Agregamos características propias de la cerveza  
    public double Alcohol { get; set; }  
    public string Style { get; set; }  
}
```

La herencia funciona igual que una clase normal, podemos utilizar lo ya existente en la clase padre, en este caso la propiedad Name y Quantity.

Una clase abstracta puede tener también métodos, tanto implementados como no implementados, estos últimos llamados métodos abstractos, y no son más que los métodos que se definen en una interface, métodos que al heredar deben implementarse.

```

abstract class Liquid
{
    public string Name { get; set; }

    public string Quantity { get; set; }

    // El método GetInfo está implementado
    public string GetInfo() => $"Líquido {Name}";

    // El método GetCompleteInfo() no está implementado
    public abstract string GetCompleteInfo();
}

```

Al heredar de la clase Liquid estamos obligados a poner funcionalidad a GetCompleteInfo():

```

class Beer : Liquid
{
    public double Alcohol { get; set; }
    public string Style { get; set; }

    // Al implementar el método abstracto debemos poner la palabra
    // override, al igual que una sobreescritura
    public override string GetCompleteInfo() =>
        $"Una cerveza {Name} con {Alcohol} de alcohol";
}

```

Clase Abstracta vs Interface

Es muy probable que al llegar a este punto te pregunte ¿Cuándo utilizar una interface y cuándo utilizar una clase abstracta?

Las clases abstractas tienen el propósito de ser plantillas de funcionamiento para no repetir código, el problema que tienen es que regularmente en la programación solo podemos heredar de una clase, pero si podemos implementar varias interfaces.

La ventaja que teníamos con las clases abstractas antes que estas si podían tener métodos implementados con funcionalidad, pero a partir de las nuevas versiones de C#, ya se puede tener interfaces con métodos con funcionalidad, pero hay algo aunque pueden tener las clases abstractas que no tienen las interfaces y es que pueden tener campos.

Una clase abstracta puede tener campos los cuales puedas reutilizar en tus clases hijas, sin necesidad de definirlos:

```
abstract class A{  
    protected string some = "info";  
}  
  
class B : A  
{  
    void Show() => Console.WriteLine(some);  
}  
  
var b = new B();  
b.Show();  
  
// Se muestra en pantalla:  
// info
```

¿Cuándo utilizar clase abstracta? Cuando exista algo que no pueda dividirse, es algo que parece fácil pero nunca es perfecto, y siempre me gusta utilizar el siguiente ejemplo:

Un triángulo es una figura de 3 lados, aquí en la tierra y en cualquier parte del universo.

Esta es una analogía de que un hecho como tal no puede cambiar, ejemplos parecidos podríamos encontrar a la hora de diseñar un sistema, a la hora de estructurar nuestro código, encontrar características abstractas que no van a cambiar o muy difícil lo harían, y si se detecta que algo puede cambiar en un futuro, ahí optaríamos por una interface que tenga solo esa parte.

Las clases abstractas pueden convivir con las interfaces, a continuación, te muestro un ejemplo:

```
// Una venta no puede existir sin un total
abstract class Sale
{
    protected decimal total;
    public Sale(decimal total) => this.total = total;
}

// No toda venta puede facturarse
interface IInvoice
{
    void CheckIn();
}

// No toda venta puede cancelarse
interface ICancel
{
    void Cancel();
}
```

En el ejemplo anterior hemos resumido aquella cosa por la cual una venta no puede existir y es un total, el cual lo hemos puesto como un campo en una clase abstracta, e identificamos qué cosas podrían estar o no en una venta que son facturar y cancelar, de esta manera abstraemos el funcionamiento ya sea en una clase abstracta e interfaces.

Una clase que utilice lo anterior podría quedar de la siguiente forma:

```
class SingleSale : Sale, IInvoice, ICancel
{
    public SingleSale(decimal total) : base(total)
    { }
```

```
public void Cancel()
{
    Console.WriteLine("Se manda a cancelar");
}

public void CheckIn()
{
    Console.WriteLine("Se manda a facturar");
}
}
```

De esta manera dejamos la posibilidad que una venta pueda ser facturable, cancelable pero nunca no tener un total, esto es parte de la abstracción que vamos ganando con la experiencia mientras más codificamos.

8. Conceptos extras, pero importantes

En esta sección voy a hablar sobre temas que considero importantes y que muchas veces son dejados de lado, y son temas que se aplican al día a día como programador.

Esta sección contiene explicaciones que pueden servirte en cualquier otro lenguaje de programación una vez entendido el concepto.

Excepciones

Cuando codificamos hay situaciones inesperadas las cuales pueden ocurrir al ejecutar nuestro programa, por ejemplo, ir por un recurso externo el cual no existe, ir a la posición no existente en un array, o querer introducir un valor invalido a un tipo de dato, este tipo de situaciones pueden controlarse con las Excepciones.

```
// Un bloque de excepción tiene la siguiente estructura
// Inicia con la palabra try
try{
    var numbers = new int[5]
    {
        1,2,3,4,5
    };

    Console.WriteLine(numbers[10]);
}
catch (Exception ex){
    // Si ocurre una excepción entrará en este apartado
    Console.WriteLine("Posición inválida del array");
}
finally
{
}
```

```
// 1. Pase error o no pase siempre se ejecutará este apartado
// 2. Este apartado es opcional
Console.WriteLine("Siempre se ejecuta esta línea");
}
```

Tendríamos el siguiente resultado:

```
// En pantalla veríamos lo siguiente:
// Posición inválida del array"
// Siempre se ejecuta esta línea
```

En el ejemplo anterior podemos ver una situación donde se accede a una posición inexistente de un array, lo cual ocasionará un error, pero gracias a las **Excepciones** podemos considerar situaciones inesperadas con nuestros programas.

Existen excepciones ya existentes en C# dependiendo situaciones en específico, esto nos da un mayor control a la hora de manejar errores, ya que podemos identificar un error dependiendo su tipo:

```
try
{
    // El método estático ReadAllText lee el texto de un archivo
    string text = File.ReadAllText("myfile.txt");
    Console.WriteLine(text);
}
catch (FileNotFoundException ex)
{
    // Podemos especificar el tipo de excepción para mayor control
    Console.WriteLine("El archivo no existe");
}
catch (Exception ex)
{
    // Podemos tener distintos tipos de excepciones
    Console.WriteLine("Ha ocurrido algo inesperado");
}
```



```
}
```

```
// En pantalla veríamos lo siguiente:  
// El archivo no existe
```

FileNotFoundException es un tipo de excepción ya existente en C#, el cual será arrojado cuando no se encuentre un archivo.

De la manera anterior tenemos mayor control en el manejo de errores, dependiendo el escenario del error actuaremos.

También podemos crear nuestras propias excepciones heredando de la clase Exception:

```
class BeerNotFoundException : Exception  
{  
    public BeerNotFoundException(string message)  
        : base(message) { }  
}
```

Y podemos utilizar nuestra excepción de la siguiente forma:

```
class Beer  
{  
    private int _quantity;  
    public int Quantity  
    {  
        get { return _quantity; }  
        set {  
            // Si el valor otorgado es menor o igual a 0, arrojamos  
            // excepción con la palabra throw más el tipo de  
            // excepción  
            if(value <= 0)  
            {  
                throw new BeerNotFoundException("No hay  
cerveza");
```

```
    }  
    _quantity = value;  
  }  
}  
}
```

La excepción que hemos creado puede ser incluida como las excepciones ya existentes de C#:

```
try  
{  
    var beer = new Beer();  
    beer.Quantity = 0;  
}  
catch (BeerNotFoundException ex)  
{  
    // ex.Message tiene el texto enviado en la excepción  
    Console.WriteLine(ex.Message);  
}  
catch (Exception ex)  
{  
    Console.WriteLine("Ha ocurrido algo inesperado");  
}  
  
// En pantalla veríamos lo siguiente:  
// No hay cerveza
```

Saber de excepciones nos aportará ayuda para hacer software que tenga mayor control y no ocasione problemáticas que el usuario no debe enterarse, y ser manejado por el mismo código o en todo caso, lanzar un mensaje de error amigable.

Serialización

La serialización en programación es convertir un objeto en un formato que permite almacenarlo o transmitirlo, esto es útil para

trabajar entre distintas aplicaciones sin importar el lenguaje de programación que utilicen.

El objetivo de la serialización es poder convertir y revertir, de esta manera podemos convertir objetos a un formato y posterior ese mismo formato convertirlo al objeto inicial siendo esto último llamado deserialización.

Dos áreas muy conocidas hoy en día en programación son el Backend y el Frontend, siendo el Frontend aplicaciones que actúan como cliente solicitando funcionalidad a servicios web que tomarían el papel de backend. Estas dos áreas tienen su base en compartir información sin importar el lenguaje de programación en el que trabajen. Esa información es serializada y deserializada.

Existen distintos formatos los cuales nos harán no reinventar la rueda, entre ellos esta XML, YAML, JSON, BSON y mucho más, pero el más utilizado hoy en día es JSON, el cual veremos a continuación.

JSON

El formato JSON, el cual sus siglas significan JavaScript Object Notation(no te dejes llevar por JavaScript) es un formato que sirve para serializar objetos y poder transmitirlos a distintas aplicaciones sin importar la tecnología.

Este formato es el más utilizado hoy en día, sobre todo en desarrollo web.

Teniendo la siguiente clase:

```
class Beer
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Creamos un objeto con la siguiente información:

```
var beer = new Beer()  
{  
  Name = "Delirium Tremens",  
  Price = 100  
};
```

Con el formato JSON su representación sería la siguiente:

```
{  
  "Name": "Delirium Tremens",  
  "Price": 100  
}
```

El texto anterior se puede guardar en un archivo, enviarlo por la red, o pasarlo a otra aplicación, y en dicho destino se puede volver a convertir a un objeto, sin importar si es JavaScript, Python, PHP etc.

El formato JSON especifica que todo objeto va entre llaves {} y cada campo va con su respectivo nombre entre comillas, los valores de cada campo están a la derecha llevando comilla solo los valores que son string o cadenas.

También podemos representar arrays, teniendo el siguiente array de números:

```
var numbers = new int[5]{  
  1,2,3,4,5  
};
```

Su representación en JSON es de la siguiente forma:

```
[1,2,3,4,5]
```

Los arrays son representados con corchetes y sus valores separados por coma.

También podemos representar array de objetos:

```
[
  {
    "Name": "Delirium Tremens",
    "Price": 100
  },
  {
    "Name": "Erdinger",
    "Price": 80
  },
  {
    "Name": "Corona",
    "Price": 20
  },
]
```

Cada objeto está envuelto en llaves `{}` y todo esto envuelto en corchetes `[]` los cuales representan que es un array.

Una vez entendido el formato JSON nos queda ver el proceso de serializar un objeto desde C# lo cual es muy sencillo:

```
// Necesitamos de una biblioteca de C#, para hacer uso basto con  
// utilizar using mas el nombre de la biblioteca  
using System.Text.Json;  
  
// La clase JsonSerializer realiza el trabajo de serialización  
string jsonBeer = JsonSerializer.Serialize(beer);  
Console.WriteLine(jsonBeer);  
  
// En pantalla se mostraría:  
// {"Name":"Delirium Tremens","Price":100}
```

El proceso inverso, teniendo un JSON, convertir en objeto se realiza igual de manera fácil así:

```
using System.Text.Json;
```

```
// con la @ al inicio especificamos que tendremos una cadena con
// comillas en ella, las cuales especificando dos veces ""a"", son
// leídas como un carácter que forma parte del contenido
string beerJson = @"{
  ""Name"":""Delirium Tremens"",
  ""Price"":100
}";

Beer myBeer = JsonSerializer.Deserialize<Beer>(beerJson);
Console.WriteLine(myBeer.Name);

// Tendríamos en pantalla:
// Delirium Tremens
```

La serialización es parte esencial para todo programador, ya que estaremos siempre trabajando con aplicaciones terceras.

9. Consejos finales

Como consejos finales, en esta sección hablaré desde mi experiencia viendo este mundo de la programación por más de dos décadas.

Desde que estoy involucrado en esto de escribir código he notado varios cambios que han pasado en cómo se hace software y al inicio era abrumador para mí, querer abarcar todas las nuevas áreas que iban naciendo, hasta que me di cuenta de que era imposible.

Los consejos generales que te daré es que no hay porque abrumarse, no podemos conocer todo en la programación, pero no por ello debemos quedarnos sin seguir aprendiendo.

Hay que aprender con objetivo, podemos curiosar con cosas que nos llamen la atención y ver si algo de eso nos serviría para un objetivo en específico, o igual, si no nos sirve no invertir más tiempo, pero saber de qué va.

Por ejemplo, cuando yo comencé a programar el concepto de framework era algo que no se escuchaba tanto, sobre todo en desarrollo web, pero a partir de la década del 2000-2010 comenzó a sonar más y más, sobre todo en empresas que solicitaban a programadores que conocieran cómo usar frameworks.

Con el tiempo fue una adopción que se ha quedado, algo que muchos de mi generación veíamos como innecesario, pero una vez te adentras te das cuenta de que te ahorran mucho tiempo, y sobre todo que están rodeados de personas magníficas que los van mejorando y comunidades que apoyan a programadores a resolver problemas en común relacionados al framework de preferencia.

Algo parecido veo ahora con la llegada de herramientas que utilizan Inteligencia Artificial, lo cual vienen a ser asistentes que puedan apoyarnos, más no reemplazarnos, pero tampoco es que sean enemigas, como lo veía yo a los Frameworks hace años, es mejor

adaptarse, y ver como pueden ser útiles para tu trabajo diario como programador.

Nacerán nuevos lenguajes de programación, saldrán nuevos frameworks, y nuevas maneras de programar, pero los conceptos se mantendrán, y espero que, al haber leído este libro, esos conceptos te sirvan, que al final, ese es el objetivo de este libro, enseñar conocimientos que son básicos y muchas veces dejados de lado.

Como programador no vas a dejar de necesitar aprender cosas nuevas, pero que eso no te impida disfrutar la vida, la vida no solo es estar laborando, la vida es un equilibrio de cosas, siempre hay que dejar tiempo para una buena cerveza, cuando no se tenga ese tiempo, es momento de pararse del escritorio y salir a disfrutar la vida.

Gracias por leer este libro, espero que la hayas pasado tan bien leyendo como yo escribiendo.

Héctor de León - 29 de julio 2023