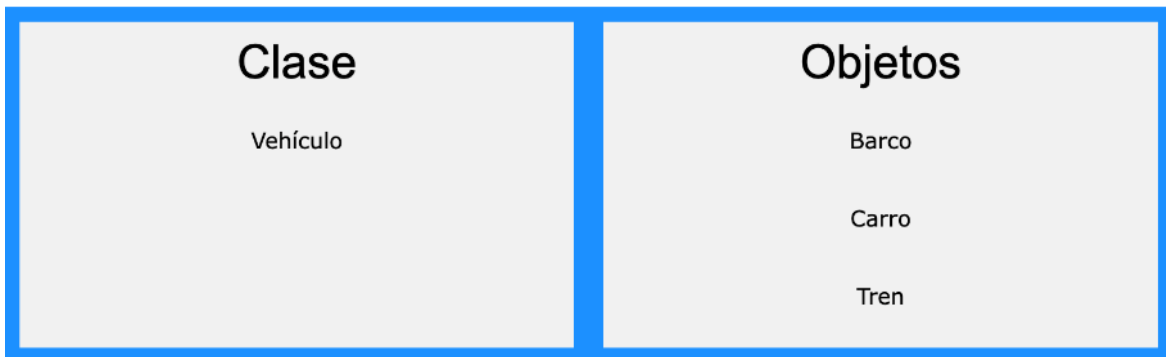


¿Qué es la Programación Orientada a Objetos (POO)?

La programación orientada a objetos es un paradigma de la programación, y su principal característica es que se basa en el uso de "objetos", estos objetos contienen información en forma de variables (usualmente se les conoce como atributos o propiedades) y código en forma de métodos (que son las acciones o procedimientos que realizan los objetos).

Cuando se trabaja con POO es importante tener clara la diferencia entre **clase** y **objeto**, debido a que no son lo mismo. Entonces, ¿Cuál es la diferencia de una clase y un objeto?

Clase	Objeto
Una clase es una especie de "plantilla" en la que se definen los atributos y métodos predeterminados de un tipo de objeto. Esta plantilla se crea para poder crear objetos fácilmente. Al método de crear nuevos objetos mediante la lectura y recuperación de los atributos y métodos de una clase se le conoce como instanciación.	Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa).



Ahora que tenemos claro que es una clase y un objeto, ¿Cómo se ven una clase y un objeto en C#?

```

public class Vehiculo
{
    //Atributos - Propiedades
    public string tipoCombustible { get; set; }
    public int capacidadPasajeros { get; set; }
    public DateTime fechaConstruccion { get; set; }

    //Métodos

    public void arrancar()
    {
        Console.WriteLine("El Vehiculo ha arrancado");
    }

    public void recargarCombustible()
    {
        Console.WriteLine("El vehículo ha recargado combustible");
    }
}

```

Ilustración 1. Ejemplo de una Clase. POO.

En la ilustración 1 se puede observar el ejemplo de una clase, cuenta con sus atributos (características que tiene un vehículo) y además podemos ver sus métodos (acciones que podría realizar) que en este ejemplo en concreto son arrancar y recargar combustible.

Ahora, solo nos falta poder ver el ejemplo de un objeto en c#, veámoslo en la siguiente ilustración:

```

1  // See https://aka.ms/new-console-template for more information
2  using P00;
3
4  Vehiculo carro = new Vehiculo();
5
6  Vehiculo barco = new Vehiculo();
7
8  Vehiculo avion = new Vehiculo();
9
10 //El obeejto carro es una instancia de la clase vehiculo
11 carro.capacidadPasajeros = 5;
12 carro.tipoCombustible = "Gasolina";
13 carro.fechaConstruccion = new DateTime(2011, 6, 10); //Año, mes, día
14
15 carro.arrancar();
16 carro.recargarCombustible();
17

```

Ilustración 2. Ejemplo de una clase. POO.

En la ilustración 2 podemos observar como se instancian 3 objetos de tipo vehiculo, y además logramos apreciar como el objeto carro es inicializado; inicializar un objeto es asignarles valor a sus atributos (variables), en este caso; la capacidad de pasajeros, tipo de combustible y la fecha de construcción fueron inicializados.

En el ejemplo anterior, pudimos ver como se pudo instanciar el objeto carro y posteriormente se inicializó, pero, ¿Cómo se puede obligar a un objeto a que se inicialice al momento de instanciarse?, esto se puede conseguir a través del uso de constructores, que son “*métodos especiales*”, este método es llamado cada vez que se instancia el objeto y nos obliga a inicializar sus atributos antes de crearlo, pero veamos un ejemplo en c# para que nos quede más claro:

```
public class Vehiculo
{
    //Atributos - Propiedades
    public string tipoCombustible { get; set; }
    public int capacidadPasajeros { get; set; }
    public DateTime fechaConstruccion { get; set; }

    //Constructor
    public Vehiculo(string combustible, int cantPasajeros, DateTime fechaConstruccion)
    {
        this.tipoCombustible = combustible;
        this.capacidadPasajeros = cantPasajeros;
        this.fechaConstruccion = fechaConstruccion;
    }
}
```

Ilustración 3. Ejemplo de constructores.

Para entender aún mejor el constructor del ejemplo anterior, vamos a explicar cada uno de sus componentes, primero debemos indicar el nivel de acceso al constructor (puede ser private o public), en este constructor elegimos el public (porque la clase también es public), escribimos Vehiculo (el mismo nombre de la clase) y dentro de los parentesis escribimos los parámetros, que son las variables que voy a utilizar para inicializar mi objeto, en este ejemplo en concreto los parámetros son combustible, cantPasajeros y fechaConstruccion, se debe tener en cuenta que estas variables deben ser del mismo tipo que los atributos, por ejemplo, el atributo tipoCombustible es de tipo *string* y el parámetro combustible también es *string*.

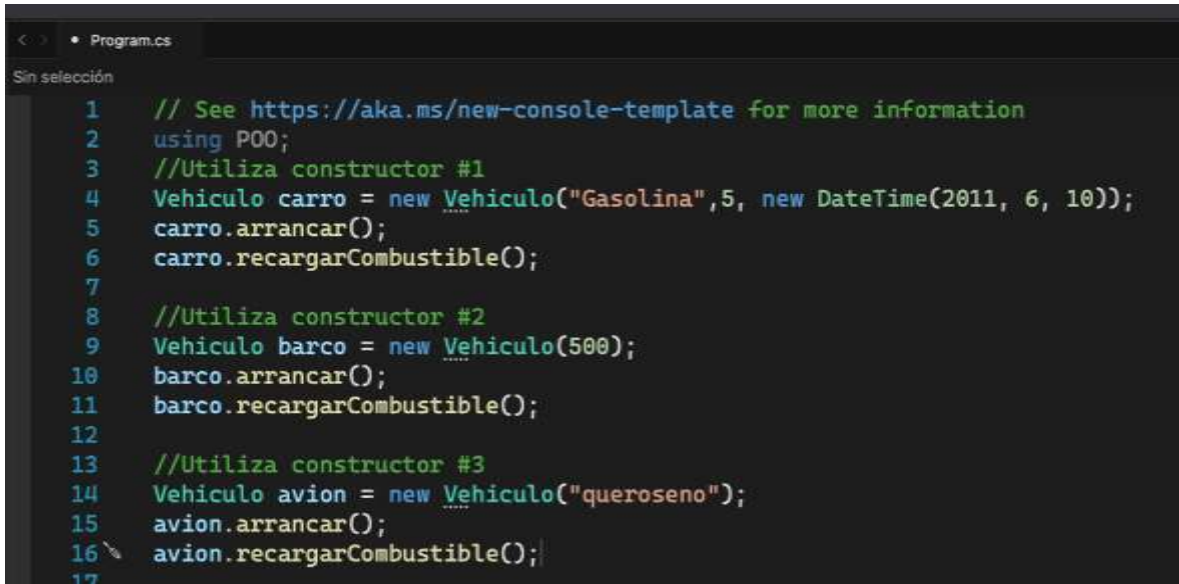
Ahora, veamos cómo se instancia una clase que tiene constructor:

```
< > x Program.cs
Sin selección
1 // See https://aka.ms/new-console-template for more information
2 using P00;
3
4 Vehiculo carro = new Vehiculo("Gasolina",5, new DateTime(2011, 6, 10));
5 carro.arrancar();
6 carro.recargarCombustible();
```

Ilustración 4. Instancia de una clase con constructor. POO

Al instanciar una clase con constructor, se deben agregar las variables de los parámetros en el mismo orden en el que se indicaron en el constructor, en este ejemplo vemos como en el constructor se indica que primero se agrega un dato de tipo string, luego un int y por ultimo un DateTime, es por esto que al instanciar esta clase las variables se deben colocar en este mismo orden de ingreso.

Dependiendo del escenario se puede requerir crear uno o mas constructores, la POO nos permite realizar esto, veamos un ejemplo más en c#:

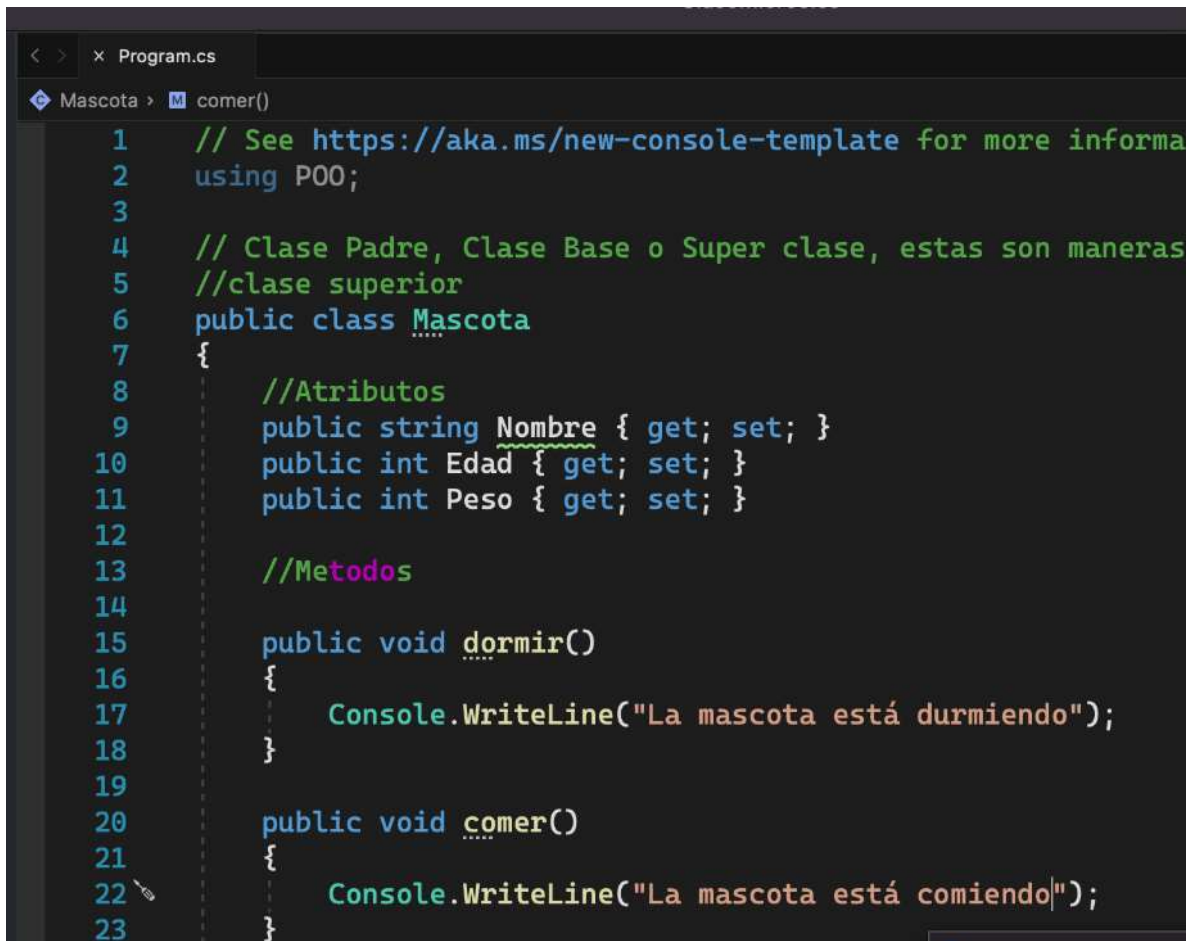


```
1 // See https://aka.ms/new-console-template for more information
2 using POO;
3 //Utiliza constructor #1
4 Vehiculo carro = new Vehiculo("Gasolina",5, new DateTime(2011, 6, 10));
5 carro.arrancar();
6 carro.recargarCombustible();
7
8 //Utiliza constructor #2
9 Vehiculo barco = new Vehiculo(500);
10 barco.arrancar();
11 barco.recargarCombustible();
12
13 //Utiliza constructor #3
14 Vehiculo avion = new Vehiculo("queroseno");
15 avion.arrancar();
16 avion.recargarCombustible();
17
```

Ilustración 5. Instanciamiento de objetos de una clase con multiple constructores. POO

En el ejemplo anterior se puede observar, que debido a que contamos con varios constructores, vamos a poder instanciar los objetos de diversas maneras.

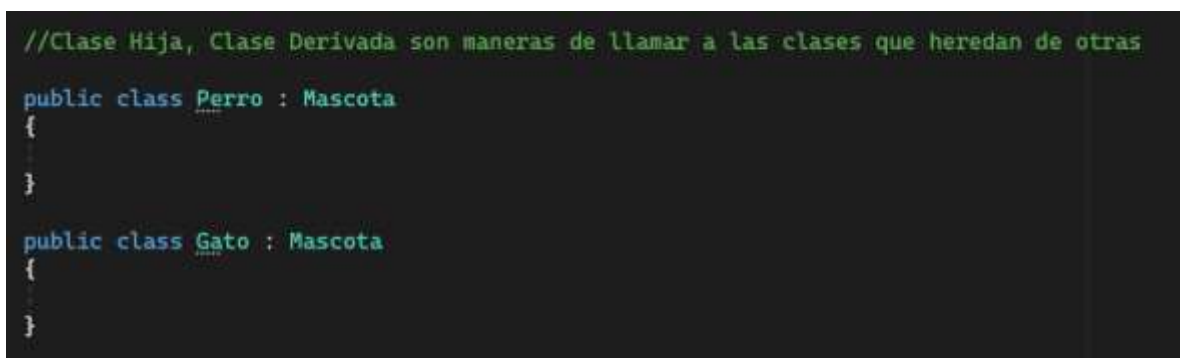
Otro tema de suma utilidad que se obtiene de la POO es la herencia, pero ¿Qué es la herencia en la POO? Es sencillamente heredar propiedades (atributos) y metodos a otras clases, esto sirve para reutilizar código (en palabras más sencillas, nos ayuda a escribir menos código y hacer mas eficiente nuestro software). Veamos otro ejemplo en c# para comprender mejor esto:



```
< > x Program.cs
Mascota > M comer()

1 // See https://aka.ms/new-console-template for more informa
2 using P00;
3
4 // Clase Padre, Clase Base o Super clase, estas son maneras
5 //clase superior
6 public class Mascota
7 {
8     //Atributos
9     public string Nombre { get; set; }
10    public int Edad { get; set; }
11    public int Peso { get; set; }
12
13    //Metodos
14
15    public void dormir()
16    {
17        Console.WriteLine("La mascota está durmiendo");
18    }
19
20    public void comer()
21    {
22        Console.WriteLine("La mascota está comiendo");
23    }
```

Ilustración 6. Clase base. POO.



```
//Clase Hija, Clase Derivada son maneras de llamar a las clases que heredan de otras

public class Perro : Mascota
{
}

public class Gato : Mascota
{
}
```

Ilustración 7. Clases hijas. POO.

Las clases Perro y Gato heredan de la clase Mascota, por eso, aunque no veamos métodos o atributos escritos en estas clases igual los contienen. Veamos otro ejemplo para tener una noción aún más clara:



```

1  // See https://aka.ms/new-console-template for more information
2  using P00;
3
4  Perro Firulais = new Perro();
5
6  Gato Misifus = new Gato();
7
8  Firulais.comer();
9  Misifus.dormir();

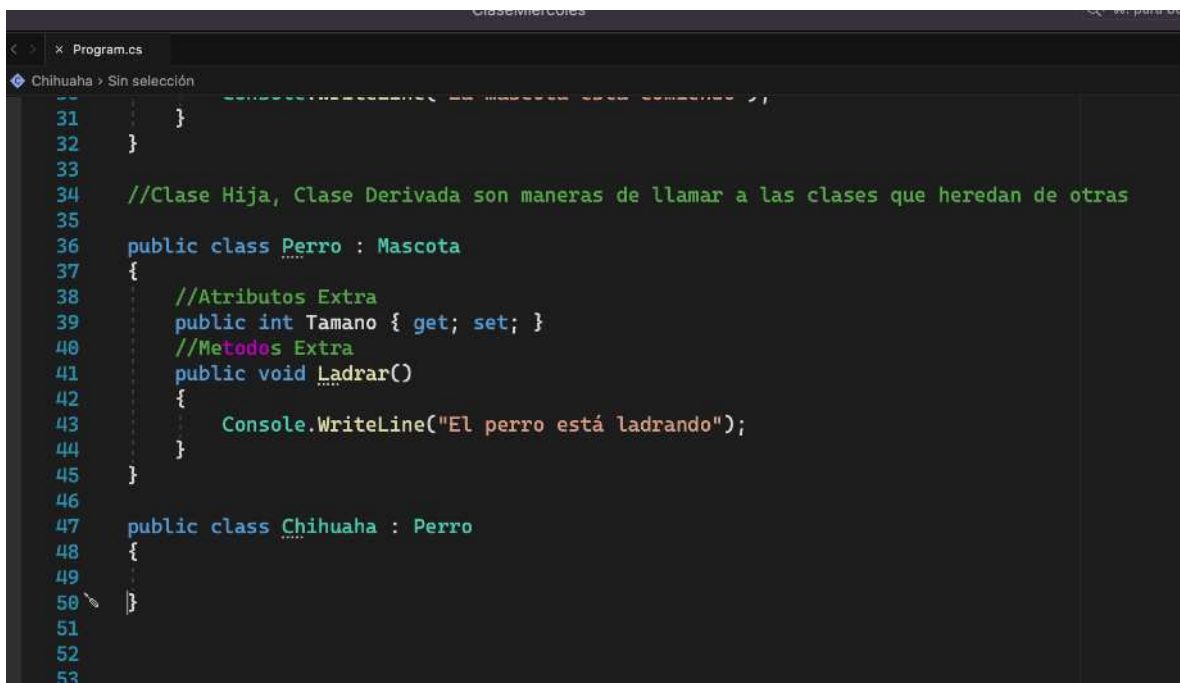
```

Ilustración 8. Ejemplo de herencias en ejecución.

En la ilustración 8 pudimos ver como los objetos de las clases Gato y Perro pueden utilizar métodos, que son heredados de la clase Mascota.

Las clases hijas también se pueden modificar, se pueden agregar más atributos y métodos según la necesidad del programa.

Además, las clases hijas se pueden convertir en clases padres, veamos el siguiente ejemplo para tener más claridad en este concepto:



```

31  }
32  }
33
34  //Clase Hija, Clase Derivada son maneras de llamar a las clases que heredan de otras
35
36  public class Perro : Mascota
37  {
38      //Atributos Extra
39      public int Tamano { get; set; }
40      //Metodos Extra
41      public void Ladrar()
42      {
43          Console.WriteLine("El perro está ladrando");
44      }
45  }
46
47  public class Chihuahua : Perro
48  {
49  }
50
51
52
53

```

Ilustración 9. Clase hija sirviendo de clase padre. POO

De igual forma, la clase chihuahua podría implementar sus propios métodos y atributos.

Este último ejemplo también sirve para hablar sobre el polimorfismo, en POO el polimorfismo es la capacidad que tienen clases heredadas de ejecutar diferentes acciones, aunque sea el mismo método, veamos la siguiente imagen para comprender mejor:

```
public class Chihuahua : Perro
{
    //Poliformismo
    public void Ladrar()
    {
        Console.WriteLine("El perro Chihuahua está ladrando muy duro");
        Console.WriteLine("El Chihuahua no quiere parar de ladrar");
    }

    //
}
```

El método “Ladrar” ejecuta diferentes acciones en Chihuahua, esto es posible debido al polimorfismo.

Otro tema muy importante de conocer en la POO son las Interfaces; que son “una clase” que contiene métodos que queremos que una o varias clases ejecuten sin obligar a las demás clases a hacerlo. Veamos otro ejemplo con código para definirlo mejor:

```
public class Perro : Mascota
{
}

public class Pajaro : Mascota
{
}

public class Gato : Mascota
{
}

public class Tortuga : Mascota
{
}
```

Ilustración 10. Clases hijas. POO.

Tenemos cuatro clases hijas: Perro, Pajaro, Gato y Tortuga; y queremos agregar los métodos “protegerCasa” y “alertarPeligro”, pero solo queremos implementarlo en las clases Perro y

Gato. Incluir un método llamado “protegerCasa” o “alertarPeligro” en la clase base Mascota no sería funcional, debido a que las clases pájaro y Tortuga también heredarían esta clase, en este escenario podríamos utilizar una Interface para solucionar este problema, veamoslo en el siguiente ejemplo:

```

62 public interface IseguridadAnimal
63 {
64     public void protegerCasa();
65
66     public void alertarPeligro();
67 }
    
```

Ilustración 11. Ejemplo de Interface. POO.

La Interfaz IseguridadAnimal nos provee los métodos protegerCasa y alertarPeligro, de esta manera podremos implementarlos unicamente en las clases deseadas. Note que el nombre de la interface lleva por delante la letra “I” en mayuscula, es una buena practica nombrar de esta manera las interfaces. En la siguiente imagen veremos la implementación de la interfaz:

```

public class Perro : Mascota, IseguridadAnimal
{
    public void alertarPeligro()
    {
        throw new NotImplementedException();
    }

    public void protegerCasa()
    {
        throw new NotImplementedException();
    }
}

public class Gato : Mascota, IseguridadAnimal
{
    public void alertarPeligro()
    {
        throw new NotImplementedException();
    }

    public void protegerCasa()
    {
        throw new NotImplementedException();
    }
}
    
```

Ilustración 12. Implementación de una interfaz.

Otro concepto que es necesario dominar en la POO es el de **Abstracción**, que lo podemos entender como una clase que solo muestra las características y comportamientos de un posible objeto, es decir, la clase abstracta sería una especie de modelo o molde abstracto que se puede utilizar para crear otras clases que aunque van a ser similares van a tener comportamientos distintos, para comprender mejor esta explicación analicemos el siguiente ejemplo:

```
//Clase abstracta Animal
public abstract class Animal
{
    public abstract void Alimentarse();

    public abstract void Cazar();

    public void ProtegerTerritorio()
    {
        Console.WriteLine("Protegiendo Territorio");
        Console.ReadKey();
    }
}
```

Ilustración 13. Ejemplo de clase abstracta. POO.

Hemos creado una clase abstracta llamada Animal, nótese la palabra “*abstract*” antes del comando class, esta clase abstracta nos permitirá crear más clases hijas que van a compartir esos mismos comportamientos (métodos) pero que van a ejecutarse de manera distinta, veamos la siguiente imagen para mayor claridad:

```
//Clases hijas - heredadas de la clase Animal

public class Leon : Animal
{
    public override void Alimentarse()
    {
        Console.WriteLine("El león se está alimentando.");
    }

    public override void Cazar()
    {
        Console.WriteLine("El león está cazando.");
    }
}

public class Elefante : Animal
{
    public override void Alimentarse()
    {
        Console.WriteLine("El león se está alimentando.");
    }

    public override void Cazar()
    {
        Console.WriteLine("Ups! Los elefantes no cazan.");
    }
}
```

Ilustración 14. Clases hijas de una clase abstracta. POO.

Como se puede observar en la ilustración anterior, las clases hijas de una abstracta pueden ejecutar distintas acciones en los métodos que heredan. La abstracción es una herramienta que nos permite poder resolver ciertos problemas que podamos enfrentar en el desarrollo de software. Se podría decir que su desventaja es que nos obliga a modificar los métodos abstractos, sin embargo, dependiendo del escenario esto nos podría ser de ayuda.

Tipos de datos en C#

En C# podemos encontrar dos tipos de datos, los **primitivos** que son tipos de datos simples (como los booleanos, int, double, etc) y tenemos también los de tipo **personalizado**, que es cuando creamos una clase, por ejemplo.

Los datos primitivos más comunes en C# son:

Nombre corto	Clase .NET	Tipo	Ancho	Intervalo (bits)
byte	Byte	Entero sin signo	8	0 a 255
sbyte	SByte	Entero con signo	8	-128 a 127
int	Int32	Entero con signo	32	-2.147.483.648 a 2.147.483.647
uint	UInt32	Entero sin signo	32	0 a 4294967295
short	Int16	Entero con signo	16	-32.768 a 32.767
ushort	UInt16	Entero sin signo	16	0 a 65535
long	Int64	Entero con signo	64	-922337203685477508 a 922337203685477507
ulong	UInt64	Entero sin signo	64	0 a 18446744073709551615
float	Single	Tipo de punto flotante de precisión simple	32	-3,402823e38 a 3,402823e38
Double	Double	Tipo de punto flotante de precisión doble	64	1,79769313486232e308 a 1,79769313486232e308
char	Char	Un carácter Unicode	16	Símbolos Unicode utilizados en el texto
bool	Boolean	Tipo Boolean lógico	8	True o false
object	Object	Tipo base de todos los otros tipos		
string	String	Una secuencia de caracteres		
decimal	Decimal	Tipo preciso fraccionario o integral, que puede representar números decimales con 29 dígitos significativos.	128	$\pm 1.0 \times 10e-28$ a $\pm 7.9 \times 10e28$

Tipos de operadores en C#

Los operadores son palabras reservadas del lenguaje que nos permiten ejecutar de operaciones lógicas en el contenido de ciertos elementos (por ejemplo, en los ciclos if, while, entre otros). Cuando combinamos uno o varios operadores y elementos con los cuales los operadores van a tomar una decisión se llama **expresión**.

Los operadores se pueden categorizar de la siguiente manera:

Operadores Aritméticos

Operador	Operación realizada	Ejemplo	Resultado
+	Suma	6+4	10
-	Sustracción	12-6	6
*	Multiplicación	3*4	12
/	División	25/3	8.3333333333
%	Módulo (resto de la división entera)	25 % 3	1

Ilustración 15. Operadores aritméticos. Fuente <https://i0.wp.com/blog.auriboxtraining.com/wp-content/uploads/2016/01/178.png>:

Operadores de Comparación

Operador	Operación realizada	Ejemplo	Resultado
=	Igualdad	2 = 5	False
!=	Desigualdad	2 <> 5	True
<	Inferior	2 < 5	True
>	Superior	2 > 5	False
<=	Inferior o igual	2 <= 5	True
>=	Superior o igual	2 >= 5	False
is	Comparación del tipo de la variable con el tipo dado	O1 is Cliente	True si la variable O1 referencia un objeto creado a partir del tipo Cliente

Ilustración 16. Operadores de comparación. Fuente: <https://i2.wp.com/blog.auriboxtraining.com/wp-content/uploads/2016/01/180.png>

Operadores lógicos

Operador	Operación	Ejemplo	Resultado
&	y Lógico	If (test1) & (test2)	verdadero si test1 y test2 es verdadero
	O lógico	If (test1) (test2)	verdadero si test1 o test2 es verdadero
^	O exclusivo	If (test1) ^ (test2)	verdadero si test1 o test2 es verdadero, pero no si los dos son verdaderos simultáneamente
!	Negación	If Not test	Invierte el resultado del test
&&	y Lógico	If (test1) && (test2)	Idem «y lógico» pero test2 sólo será evaluado si test1 es verdadero
	O lógico	If (test1) (test2)	Idem «o lógico» pero test2 sólo será evaluado si test1 es falso

Ilustración 17. Operadores lógicos. Fuente: <https://i1.wp.com/blog.auriboxtraining.com/wp-content/uploads/2016/01/181.png?w=719&ssl=1>

Tipos de ciclos en C#

Podemos decir que un ciclo en la programación es un conjunto de instrucciones que se van a ejecutar “x” cantidad de veces, y estas instrucciones se van a repetir hasta que la condición asignada a dicho ciclo deje de cumplirse. Usualmente los ciclos se usan para repetir una o un grupo de instrucciones sin tener que escribir varias veces el mismo código, con esto nos ahorramos tiempo, esfuerzo y permite que nuestro código quede más claro. Veamos cuales son los ciclos más comunes en C#:

- **FOR:** El ciclo **for** ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true. En el ejemplo siguiente se muestra la instrucción **for**, que ejecuta su cuerpo mientras que un contador entero sea menor que tres:

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
```

- **FOREACH:** El ciclo **foreach** ejecuta una instrucción o un bloque de instrucciones para cada elemento dentro de una lista, array, etc. Es decir, itera (recorre) cada uno de los elementos que componen a un objeto que contiene más objetos.

```
int[] Numeros = {1,2,3,4,5,6,7,8};

foreach (int item in Numeros)
{
    System.Console.WriteLine(item);
}
```

En el ejemplo anterior, el ciclo **foreach** lo que hace es imprimir cada variable de tipo **int** que contiene el **array** *Números*.

- **WHILE:** Este tipo de ciclo se utiliza cuando tienes que realizar una acción basado en una condición, por ejemplo, pensemos que tenemos un balón de básquet, y mientras no encestemos, tendremos que seguir intentando encestar, en este caso el ciclo se repetirá mientras la condición sea válida (en este ejemplo mientras no enceste)

```
bool Encestar = false;

while (false)
{
    System.Console.WriteLine("Seguir lanzando pelota");
}
```

- **DO:** Este tipo de ciclo es similar al **while**, sin embargo, este ciclo en su primera ejecución va a realizar una acción, sin importar si la misma se cumple o no, es hasta la segunda ejecución que valida si la condición se cumple o no.

```
bool Encestar = false;
do
{
    System.Console.WriteLine("Seguir lanzando pelota");
}while(Encestar == false)
```


Operaciones de tipo IF ELSE en C#

Las instrucciones IF son simplemente un bloque de código que nos permitirá ejecutar una acción siempre y cuando se cumpla una condición. Podemos decir que son bloques para toma de decisiones, veamos el siguiente ejemplo para tener más claro el concepto:

```
int Validador = 3;  
if(Validador>2)  
{  
  
    System.Console.WriteLine("El validador es mayor a 2");  
  
}else{  
  
    System.Console.WriteLine("El validador no es mayor a 2");  
  
}
```

En el ejemplo anterior, una variable de tipo **int** es inicializada con el valor de 3, si el validador es mayor a 3 el programa va a imprimir en pantalla el texto indicando que es mayor a 2, caso **contrario**, va a imprimir que el validador no es mayor a 2.