Guía Teórica – Curso .Net con C#

# Conceptos Básicos

## Entry Point

El punto de entrada para los programas en C#

```
class Program
    {
        static void Main(string[] args)
       {
       Console.WriteLine("Hola Mundo");
       }
    }
```

static: Es un modificador que permite ejecutar un método sin tener que instanciar a una variable (sin crear un objeto). El método Main() debe ser estático.

void: Indica el tipo de valor de retorno del método Main(). No necesariamente tiene que ser void.

string [] args: Es un Array de tipo string que puede recibir el método Main() como parámetro. Este parámetro es opcional.

## Programación Orientada a Objetos

- Es una manera de construir Software basada en un nuevo paradigma.
- Propone resolver problemas de la realidad a través de identificar objetos y relaciones de colaboración entre ellos.
- El Objeto y el Mensaje son sus elementos fundamentales.

Características:

1. Herencia: es un tipo de relación entre clases. Va de la generalización a la especialización. Se hereda la implementación.
2. Polimorfismo: la definición del método reside en la clase padre o base. La implementación del método reside en la clase hija o derivada.
3. Encapsulamiento: denota la capacidad del objeto de responder a peticiones a través de sus métodos o propiedades sin la necesidad de exponer los medios utilizados para llegar a brindar estos resultados.
4. Abstracción: Ignorancia selectiva / Se enfoca en lo que es importante

## Clases

Una clase es una clasificación. Clasificamos en base a comportamientos y atributos comunes.
Es una abstracción de un objeto.

Sintaxis

```
[modificador] class Identificador
 {
 }
```

modificador: Determina la accesibilidad que tendrán sobre ella otras clases.
class: Es una palabra reservada que le indica al compilador que el siguiente código es una clase.
Identificador: Indica el nombre de la clase.

| Nombre | Descripción |
|---|---|
| Abstract | Indica que la clase no podrá instanciarse |
| Internal | Accesible en todo el proyecto |
| Public | Accesible desde cualquier proyecto |
| Private | Accesor por defecto. |
| Sealed | Indica que la clase no podra heredar |
| | |

Atributos

```
[modificador] tipo identificador;
```

modificador: Determina la accesibilidad que tendrán sobre él las demás clases. Por defecto es private.
tipo: Representa al tipo de dato. Ejemplo: int, float, etc.
Identificador: Indica el nombre del atributo.

| Nombre | Puede ser accedido por… |
|---|---|
| Private | Los miembros de la misma clase |
| Protected | Los miembros de la misma clase y clases derivadas o hijas |
| Internal | Los miembros del mismo proyecto |
| Public | Cualquier miembro. |
| | |

Métodos

```
[modificador] retorno Identificador([args])
        {
            //sentencias
        }
```

modificador: Determina la forma en que los métodos serán usados.
retorno: Es el tipo de valor devuelto por el método (sólo retornán un único valor).
Identificador: Indica el nombre del método.
args: Representan una lista de variables cuyos valores son pasados al método para ser usados por este. Los corchetes indican que los parámetros son opcionales.
Si un método no retorna ningún valor se usará la palabra reservada void.
Para retornar algún valor del método se utilizará la palabra reservada return.
Los parámetros se definen como:

```
tipoDato identificador_parametro
```

Modificadores:

| Nombre | Descripción |
|---|---|

| | |
|---|---|
| Abstract | Sólo la firma del método, sin implementar. |
| public | Accesible desde cualquier proyecto. |
| private | Sólo accesible desde la clase. |
| protected | Sólo accesible desde la clase o derivadas. |
| static | Indica que es un método de clase. |
| virtual | Permite definir métodos, con su implementación, que podrán ser sobrescritos en clases derivadas. |

Namespaces

- Es una agrupación lógica de clases y otros elementos.
- Toda clase esta dentro de un NameSpace.
- Proporcionan un marco de trabajo jerárquico sobre el cuál se construye y organiza todo el código.
- Su función principal es la organización del código para reducir los conflictos entre nombres.
- Esto hace posible utilizar en un mismo programa componentes de distinta procedencia.

Directivas:

Son elementos que permiten a un programa identificar los NameSpaces que se usarán en el mismo. Permiten el uso de los miembros de un NameSpace sin tener que especificar un nombre completamente cualificado.
Directivas de un namespace:

1. using
2. alias

Los namespaces pueden contener:

- Clases
- Delegados
- Enumeraciones
- Interfaces
- Estructuras
- Namespaces
- Directivas using y alias

```
using System; //directiva using

using SC = System.Console; //directiva alias
```

Creación de un namespace:

```
namespace Identificador{

}
```

Constantes

Una constante contiene un valor que se asigna cuando se compila el programa y nunca cambia.

```
const int cantidadDePatas = 4
```

## Objetos

Los objetos son instancias de una clase, son creados en tiempo de ejecución y poseen comportamiento (métodos) y estado (atributos).
Para acceder tanto a los atributos como a los métodos se utiliza el operador punto ( . ).
Para crear un objeto se necesita la palabra reservada NEW.

```
NombreClase nombre_objeto = new NombreClase();
```

NombreClase: Es el identificador de la clase o del tipo de objeto al que se referirá.
nombre_objeto: Es el nombre asignado a la instancia de tipo Nombre_Clase.
NombreClase(): Es el constructor del objeto y no el tipo de objeto.
Una vez inicializado el objeto se puede utilizar para manipular sus atributos y llamar a sus métodos.

Aunque no se escriba ningún constructor, existe uno por defecto que se usa cuando se crea un objeto a partir de un tipo referencia.
Los constructores llevan el mismo nombre de la clase.

Hay dos tipos de constructores:

- Constructores de instancia: que inicializan objetos (atributos no estaticos)
- Constructores estaticos: que inicializan clases (atributos estaticos)

Caracteristicas de un constructor por defecto:

1. Acceso publico
2. Mismo nombre que la clase
3. No tiene tipo de retorno
4. No recibe ningun argumento
5. Inicializa todos los campos a cero, null o false

```
Class MiClase
{
        public MiClase()
        {
        }
}
```

Si el constructor por defecto generado por el compilador no es adecuado, lo mejor es que escribamos nuestro propio constructor.

Constructores Estáticos

- Solo inicializará los atributos estáticos.
- No debe llevar modificadores de acceso (public, private)
- Utilizan la palabra reservada static.
- No pueden recibir parámetros.

```
Class MiClase
{
	public static int num1;
	public static int num2;

	static MiClase()
	{
		num1 = 1;
		num2 = 5;
	}
}
```

## Sobrecarga de Métodos

Los métodos no pueden tener el mismo nombre que otros elementos en una misma clase (atributos, propiedades, etc.). Sin embargo, dos o más métodos en una clase sí pueden compartir el mismo nombre. A esto se le da el nombre de sobrecarga.
Los métodos se sobrecargan cambiando el número,  el tipo y el orden de los parámetros (se cambia la firma del método).
El compilador de C# distingue métodos sobrecargados comparando las listas de parámetros.

```
Class Sobrecarga
{
	static int Sumar(int a, int b)
	{
		return a+b;
	}

	static int Sumar(int a, int b, int c)
	{
		return a+b+c;
	}

	static void Main()
	{
		Console.WriteLine(Sumar(1,2) +
Sumar(1,2,3));
	}
}
```

Las firmas de los métodos deben ser únicas dentro de una clase.

Forman la definición de la firma de un método:

- Nombre del método.
- Tipo de parámetros.
- Cantidad de parámetros.
- Modificador de parámetro (out o ref)

No afectan la firma de un método:

- Nombres de parámetros.
- Tipo de retorno del método.

## Sobrecargar de Constructores

Al igual que los métodos, los constructores también se pueden sobrecargar.
Las normas para hacerlo son las mismas.
Se suele hacer cuando se quiere dar la posibilidad de instanciar objetos de formas diferentes.

```
Class Sobrecarga
{
	public Sobrecarga()
	{
		...
	}

	public Sobrecarga(int a)
	{
		...
	}

	public Sobrecarga(int a, int b)
	{
		...
	}

	public Sobrecarga(string mensaje)
	{
		...
	}

}
```

## Sobrecarga de Operadores

Sobrecargar un operador consiste en modificar su comportamiento cuando este se utiliza con una determinada clase.

Sintaxis de un operador sobrecargado:

```
[acceso] static TipoRetorno operator nombreOperador(tipo a[,tipo b])
{
	...
}
```

| Operadores sobrecargables | Tipo de Operadores |
|---|---|
| +, -, !, ~, ++, - -, true, false | Unarios |
| +, -, *, /, %, &, \|, ^, <<, >> | Binarios |
| **==, !=, <, >, <=, >=** | Comparación * |

* Los operadores de Comparación, si son sobrecargados, se deben sobrecargar en pares; es decir, si se sobrecarga el operador ==, se deberá sobrecargar el operador !=.

Ejemplo:

Se quiere sumar una cierta cantidad de metros con otra cantidad de centímetros.

```
double metros = 10;
double centimetros = 10;

double sumaMetros = metros + centimetros;
double sumaCentimetros = centimetros + metros;

Console.WriteLine(sumaMetros);
Console.WriteLine(sumaCentimetros);
```

Los resultados esperados serian:
SumaMetros = 10.1 (metros)
SumaCentimetros = 1010 (centimetros)

Pero recibimos:
SumaMetros = 20
SumaCentimetros = 20

Posible solución:
Crear una clase Metro y otra Centimetro y sobrecargar en ambas el operador + para realizar la suma correcta sin importar las unidades que se sumen.

Clase Metro:

```
Public class Metro
{
	public double cantidad;

	public Metro() //constructores
	{}

	public Metro(double cant)
	{
		this.cantidad = cant;
	}

	//sobrecarga operador +
	public static Metro operator +(Metro m, Centimetro c)
	{
		Metro rta = new Metro();
		rta.cantidad = m.cantidad + c.cantidad/100;
		return rta;
	}
}
```

Clase Centimetro:

```
Public class Centimetro
{
	public double cantidad;

	public Centimetro() //constructores
	{}

	public Centimetro(double cant)
	{
		this.cantidad = cant;
	}

	//sobrecarga operador +
	public static Centimetro operator +(Centimetro c, Metro m)
	{
		Centimetro rta = new Centimetro();
		rta.cantidad = c.cantidad + m.cantidad*100;
		return rta;
	}
}
```

Main:

```
Metro metros = new Metro(10);
Centimetro centimetros = new Centimetro(10);

Metro sumaMetros = metros + centimetros;
Centimetro sumaCentimetros = centimetros + metros;

Console.WriteLine(sumaMetros.cantidad);
Console.WriteLine(sumaCentimetros.cantidad);
```

Las conversiones definidas permiten hacer compatibles tipos que antes no lo eran.
Los operadores de conversión pueden ser implícitos o explícitos.
Los operadores de conversión implícitos son más fáciles de usar, aunque los operadores de conversión explícitos son muy usados cuando se quiere que los usuarios estén conscientes que una conversión se llevará a cabo.

Sintaxis:

```
public static implicit operator nombreTipo (Tipo a)
{
	...
}

public static explicit operator nombreTipo (Tipo a)
{
	...
}
```

Ejemplo con operador explicito:

Clase Metro:

```
Public class Metro
{
	public double cantidad;

	//constructores y métodos

	public static explicit operator Metro(double cant)
	{
		Metro rta = new Metro(cant);
		return rta;
	}

	public static explicit operator double(Metro m)
	{
		return m.cantidad;
	}
}
```

Clase Centimetro:

```
public class Centimetro
{
	public double cantidad;

	//constructores y métodos

	public static explicit operator Centimetro(double cant)
	{
		Centimetro rta = new Centimetro(cant);
		return rta;
	}

	public static explicit operator double(Centimetro m)
	{
		return m.cantidad;
	}
}
```

Main:

```
Metro metros = (Metro)10;
Centimetro centimetros = (Centimetro)10;

Metro sumaMetros = metros + centimetros;
Centimetro sumaCentimetros = centimetros + metros;

Console.WriteLine((double)sumaMetros);
Console.WriteLine((double)sumaCentimetros);
```

Beneficios:

- No se utilizan constructores
- No se utilizan atributos
- Código más fácil de entender

Ejemplo con Implicit:

Si se coloca implicit en vez de explicit, en la sobrecarga de los operadores de conversión, el Main sería el siguiente:

```
Metro metros = 10;
Centimetro centimetros = 10;

Metro sumaMetros = metros + centimetros;
Centimetro sumaCentimetros = centimetros + metros;

Console.WriteLine(sumaMetros);
Console.WriteLine(sumaCentimetros);
```