# Conceptos de Orientación de Objetos

## Clase

Es una colección de objetos.

## Objeto

Es una entidad en tiempo real.

Un objeto puede considerarse una "cosa" que puede realizar un conjunto de actividades relacionadas. El conjunto de actividades que realiza el objeto define el comportamiento del objeto. Por ejemplo, una persona (objeto) puede correr o saltar. En términos POO puros, un objeto es una instancia de una clase.

El siguiente ejemplo describe la clase Persona.

| **Persona** |
| --- |
| - nombre: string;<br>- edad: int; |
| + Persona (string n, int e);<br>+ Correr();<br>+ Saltar (); |

La clase se compone de tres cosas: nombre, atributos y métodos

```
public class persona {}

persona objetoPersona = new persona();
```

En el ejemplo anterior podemos decir que el objeto **Persona**, llamado **objetoPersona**, se ha creado fuera de la clase de personas.

En el mundo real, a menudo encontrarás muchos objetos individuales del mismo tipo. Por ejemplo, puede haber miles de motos en existencia, todas de la misma marca y modelo. Cada moto se ha construido a partir del mismo plano de diseño. En términos orientados a objetos, decimos que la **moto** es una instancia de la clase de objetos conocidos como **motos**.

## La encapsulación

La encapsulación es un proceso de enlace de los miembros de datos y las funciones de los miembros en una sola unidad.

Un ejemplo de encapsulación es la clase. Una clase puede contener estructuras de datos y métodos.

Veamos la siguiente clase:

```
public class EjemploEncapsulacion
{
    public class Apertura
    {

        public Apertura()
        {
        }

        double altura;
        double ancho;
        double grosor;

        public double obtener_volumen()
        {
            Console.WriteLine("Introduce la Altura");
            altura = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Introduce la Ancho");
            ancho = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Introduce la Grosor");
            grosor = Convert.ToDouble(Console.ReadLine());
            double volumen = altura * ancho * grosor;
            if (volumen < 0)
                return 0;
            return volumen;
        }
    }


    public static void Main()
    {
        double volumen;
        Apertura a = new Apertura();
        volumen = a.obtener_volumen();
        Console.WriteLine(volumen);
        Console.ReadKey();
    }


}
```

En este ejemplo, resumimos algunos datos como altura, ancho, grosor y el método **obtener_volumen()**. Otros métodos u objetos podrán interactuar con este objeto a través de métodos que tengan un modificador de acceso público

## Abstracción

La abstracción es un proceso de ocultar los detalles de la implementación y mostrar las características esenciales.

Por Ejemplo: una computadora portátil consta de muchas cosas, como procesador, placa base, RAM, teclado, pantalla, antena inalámbrica, cámara web, puertos USB, batería, altavoces, etc. Para usarlo, no necesitas saber cómo funcionan las pantallas LCD internas, teclado, cámara web, batería, antena inalámbrica, trabajos de altavoz. Solo necesita saber cómo operar la computadora portátil. Imagínate si los usuarios tuviesen que conocer todos los detalles internos de la computadora portátil antes de poder trabajar con ella. Esto tendría un costo muy elevado y no sería fácil que los usuarios pudiesen usarlas.

Así que aquí el portátil es un objeto que está diseñado para ocultar su complejidad.

Para poder abstraer usamos los especificadores de acceso

- **Public**: accesible fuera de la clase a través de referencia de objeto.
- **Private**: accesible dentro de la clase solo a través de funciones miembro.
- **Protected**: Al igual que privado pero accesible en clases derivadas también a través de funciones miembro.
- **Internal**: Visible dentro del conjunto. Accesible a través de los objetos.
- **Protected internal**: visible dentro del ensamblaje a través de objetos y en clases derivadas fuera del ensamblaje a través de funciones miembro.

Veamos un ejemplo práctico:

```
using System;
using System.Runtime.InteropServices;

public class EjemploAbstraccion
{
    public class Clase1
    {
        //Si no especifica Acceso, es privado
        int i;

        // Publico
        public int j;

        //Protected
        protected int k;

        // Internal quiere decir que es visible dentro del ensamblado
        internal int m;

        // Acceso desde ensamblaje interno así como desde clases derivadas fuera
de ensamblaje
        protected internal int n;

        // También private
        static int x;

        //Static significa compartido entre objetos
```

```
        public static int y;

        //externo quiere decir que ha sido declarado en este ensamblado definido
en algún otro ensamblado
        [DllImport("MiDll.dll")]
        public static extern int MyFoo();
        public void metodo()
        {
            //Dentro de una clase si has creado un objeto de la misma clase
entonces puedes acceder a todos los miembros a través de la referencia al objeto
incluso si los datos son private
            Clase1 objeto = new Clase1();
            //Aquí los datos son accesibles
            objeto.i = 10;
            objeto.j = 10;
            objeto.k = 10;
            objeto.m = 10;
            objeto.n = 10;
        //  objeto.x =10;  //Error los datos Static pueden ser accedidos solo por
el nombre de la clase
            Clase1.x = 10;
         // objeto.y = 10; // Error los datos Static pueden ser accedidos solo
por el nombre de la clase
            Clase1.y = 10;
        }
    }
        public static void Main()
            {

            }


}
```

Ahora si accedemos a los mismos miembros desde dentro del **Main** e intentamos compilar:

```
using System;
using System.Runtime.InteropServices;

public class EjemploAbstraccion
{
    public class Clase1
    {
        //Si no especifica Acceso, es privado
        int i;

        // Publico
        public int j;

        //Protected
        protected int k;

        // Internal quiere decir que es visible dentro del ensamblado
        internal int m;

        // Acceso desde ensamblaje interno así como desde clases derivadas fuera
de ensamblaje
        protected internal int n;
```

```
        // También private
        static int x;

        //Static significa compartido entre objetos
        public static int y;

        //externo quiere decir que ha sido declarado en este ensamblado definido
en algún otro ensamblado
        [DllImport("MiDll.dll")]
        public static extern int MyFoo();
        public void metodo()
        {
            //Dentro de una clase si has creado un objeto de la misma clase
entonces puedes acceder a todos los miembros a través de la referencia al objeto
incluso si los datos son private
            Clase1 objeto = new Clase1();
            //Aquí los datos son accesibles
            objeto.i = 10;
            objeto.j = 10;
            objeto.k = 10;
            objeto.m = 10;
            objeto.n = 10;
        //  objeto.x =10;  //Error los datos Static pueden ser accedidos solo por
el nombre de la clase
            Clase1.x = 10;
         // objeto.y = 10; // Error los datos Static pueden ser accedidos solo
por el nombre de la clase
            Clase1.y = 10;
        }
    }
public static void Main()
  {

        Clase1 objeto = new Clase1();

    //  objeto.i = 10;   // Error debido al nivel de protección - private
        objeto.j = 20;
    //  objeto.k = 30;  // Error debido al nivel de protección - static
        objeto.m = 40;
        objeto.n = 50;

        Console.WriteLine(objeto.j);
        Console.WriteLine(objeto.m);
        Console.WriteLine(objeto.n);

        Console.ReadKey();

    }

}
```

¿Qué pasa si **Main** está dentro de otro ensamblaje?

```
using System;
using System.Runtime.InteropServices;

public class EjemploAbstraccion
{
    public class Clase1
    {
        //Si no especifica Acceso, es privado
        int i;

        // Publico
        public int j;

        //Protected
        protected int k;

        // Internal quiere decir que es visible dentro del ensamblado
        internal int m;

        // Acceso desde ensamblaje interno así como desde clases derivadas fuera
de ensamblaje
        protected internal int n;

        // También private
        static int x;

        //Static significa compartido entre objetos
        public static int y;

        //externo quiere decir que ha sido declarado en este ensamblado definido
en algún otro ensamblado
        [DllImport("MiDll.dll")]
        public static extern int MyFoo();
        public void metodo()
        {
            //Dentro de una clase si has creado un objeto de la misma clase
entonces puedes acceder a todos los miembros a través de la referencia al objeto
incluso si los datos son private
            Clase1 objeto = new Clase1();
            //Aquí los datos son accesibles
            objeto.i = 10;
            objeto.j = 10;
            objeto.k = 10;
            objeto.m = 10;
            objeto.n = 10;
        //  objeto.s =10;  //Error los datos Static pueden ser accedidos solo por
el nombre de la clase
            Clase1.x = 10;
         // objeto.y = 10; // Error los datos Static pueden ser accedidos solo
por el nombre de la clase
            Clase1.y = 10;
        }
    }
        public static void Main()
            {

        Clase1 objeto = new Clase1();

    //  objeto.i = 10;   // Error debido al nivel de protección
        objeto.j = 20;
```

```
        //  objeto.k = 30;  // Error debido al nivel de protección
        //  objeto.m = 40;  // Error debido a que no puede acceder a los datos
internos fuera del ensamblaje
        //  objeto.n = 50;  // Error debido a que no puede acceder a los datos
internos fuera del ensamblaje
        //  objeto.s =10;  // Error debido a que solo se puede acceder a los
datos Static por nombres de clase
        //  Clase1.x = 10;  // Error debido a que no se puede acceder a datos
privados fuera de clase
        //  objeto.y = 10; // Error debido a que solo se puede acceder a los
datos Static por nombres de clase
        Clase1.y = 10;


        Console.WriteLine(objeto.j);
        Console.WriteLine(Clase1.y);

        Console.ReadKey();

    }


}
```

En la programación orientada a objetos, la complejidad se gestiona utilizando la abstracción. La abstracción es un proceso que implica identificar el comportamiento crítico de un objeto y eliminar detalles irrelevantes y complejos.

## Herencia

La herencia es el proceso de derivar la nueva clase de una clase ya existente.

C # es un lenguaje de programación orientado a objetos. La herencia es uno de los conceptos principales de la programación orientada a objetos. Te permite reutilizar el código existente. A través del uso efectivo de la herencia, puedes ahorrar mucho tiempo en la programación y también reducir los errores, lo que a su vez aumentará la calidad del trabajo y la productividad. Un ejemplo simple para entender la herencia en C #.

```
using System;

public class EjemploHerencia
{
    public class ClaseBase
    {
        public ClaseBase()
        {
            Console.WriteLine("Constructor de la Clase Base ejecutado");
        }

        public void Mostrar()
        {
            Console.WriteLine("Método Escritura de la Clase Base ejecutado");
        }
```

```
    }

    public class ClaseHija : ClaseBase
    {

        public ClaseHija()
        {
            Console.WriteLine("Constructor de la Clase Hija ejecutado ");
        }

        public static void Main()
        {
            ClaseHija CH = new ClaseHija();
            CH.Mostrar();
            Console.ReadKey();
        }
    }

}
```

**Resultado**:

```
Constructor de la Clase Base ejecutado

Constructor de la Clase Hija ejecutado

Método Escritura de la Clase Base ejecutado
```

En el método **Main ()** en **ClaseHija** creamos una instancia de **ClaseHija**. Entonces llamamos al método **Mostrar()**. La clase **ClaseHija** no tiene un método **Mostrar()**. Este método **Mostrar()** ha sido heredado de la clase principal **ClaseBase**.

Esta salida prueba que cuando creamos una instancia de una clase secundaria, se llamará automáticamente al constructor de la clase base antes que al constructor de la clase secundaria. Por lo tanto, en general, las clases base se instancian automáticamente antes que las clases derivadas.

En C #, la sintaxis para especificar la relación entre la clase **ClaseBase** y la clase **ClaseHija** se especifica agregando dos puntos, **":"**, después del identificador de clase derivado y luego especificando el nombre de la clase base.

**Sintaxis**:

```
class ClaseHija : ClaseBase
{
    // Código

}
```

C # solo admite la herencia de una sola clase. Lo que esto significa es que su clase puede heredar de una sola clase base a la vez. En el siguiente ejemplo, la clase C está intentando heredar de la clase A y B al mismo tiempo. Esto no está permitido en C # y nos provocará un error de tiempo de compilación.

```
public class A
{
}
public class B
{
}

public class C : A
{

}
```

En C # la herencia anidada es posible. En el siguiente ejemplo podemos ver la herencia de anidada. La clase B se deriva de la clase A. La clase C se deriva de la clase B. Por lo tanto, la clase C tendrá acceso a todos los miembros presentes tanto en la clase A como en la clase B. Como resultado de la herencia de múltiples niveles, la clase tiene acceso a MetodoA () , MetodoB () y MetodoC ().

Nota: Las clases pueden heredar de múltiples interfaces al mismo tiempo. Pregunta de la entrevista: ¿Cómo se puede implementar la herencia múltiple en C #? Respuesta: Usando Interfaces. Hablaremos de interfaces en nuestro último artículo.

```
using System;

public class EjemploHerencia
{
    public class A
    {
        public void MetodoA()
        {
            Console.WriteLine("Llamando al Método de la Clase A");
        }
    }
    public class B : A
    {
        public void MetodoB()
        {
            Console.WriteLine("Llamando al Método de la Clase B");
        }
    }
    public class C : B
    {
        public void MetodoC()
        {
            Console.WriteLine("Llamando al Método de la Clase C");
        }

        public static void Main()
        {
            C C1 = new C();
            C1.MetodoA();
            C1.MetodoB();
            C1.MetodoC();
```

```
            Console.ReadKey();
        }
    }

}
```

**Resultado**:

```
Llamando al Método de la Clase A

Llamando al Método de la Clase B

Llamando al Método de la Clase C
```

Cuando deriva una clase de una clase base, la clase derivada heredará todos los miembros de la clase base excepto los constructores. En el siguiente ejemplo la clase B, heredará METODO1 y METODO2 de la clase A, pero no puede acceder a METODO2 debido al modificador de acceso privado. Se puede acceder a los miembros de la clase declarados con un modificador de acceso privado solo en la clase. Pero, ¿Se heredan los miembros de la clase privada a la clase derivada? Sí, los miembros privados también se heredan en la clase derivada pero no podremos acceder a ellos. Al intentar acceder a un miembro privado de la clase base en la clase derivada se reportará un error de tiempo de compilación.

```
using System;

public class EjemploHerencia
{
    public class A
    {
        public void METODO1()
        {
        }
        private void METODO2()
        {
        }
    }


    public class B : A
    {
        public static void Main()
        {
            B B1 = new B();
            B1.METODO1();

       //Error, no se puede acceder al método privado METODO2
          //B1.METODO2 ();
        }
    }

}
```

A continuación, vamos a ver cómo podemos ocultar un método en C #. La clase principal tiene un método **Mostrar()** que está disponible para la clase secundaria. En la clase secundaria se ha creado un nuevo método **Mostrar()**. Entonces, ahora si creamos

una instancia de la clase secundaria y llamamos al método **Mostrar()**, se llamará al método **Mostrar()** de la clase secundaria. La clase secundaria está ocultando el método **Mostrar()** de la clase base. Esto se llama ocultación del método.

Si queremos llamar al método **Mostrar()** de la clase principal, tendríamos que escribir **cast** el objeto secundario al tipo **Parent** y luego llamar al método **Mostrar()** como se muestra en el fragmento de código a continuación.

```
using System;

public class EjemploHerencia
{
    public class Padre
    {
        public void Mostrar()
        {
            Console.WriteLine("Método Mostrar de la Clase Padre");
        }
    }

    public class Hijo : Padre
    {
        public new void Mostrar()
        {
            Console.WriteLine("Método Mostrar de la Clase Hijo");
        }

        public static void Main()
        {
            Hijo C1 = new Hijo();
            C1.Mostrar();
            //hacemos casting para que sea del tipo Padre y llame al método
Mostrar()
            ((Padre)C1).Mostrar();
            Console.ReadKey();
        }
    }

}
```

**Resultado**:

```
Método Mostrar de la Clase Padre

Método Mostrar de la Clase Hijo
```

## Polimorfismo

Cuando un método puede ser procesado de diferentes maneras se llama polimorfismo. El polimorfismo significa muchas formas. El polimorfismo es uno de los conceptos fundamentales de la POO.

El polimorfismo proporciona las siguientes características:

- Le permite invocar métodos de clase derivada a través de referencias a la clase base durante el tiempo de ejecución.
- Tiene la capacidad para que las clases proporcionen diferentes implementaciones de métodos que se llaman a través del mismo nombre.

El polimorfismo es de dos tipos:

- Polimorfismo de tiempo de compilación (sobrecarga/overload)
- Polimorfismo en tiempo de ejecución (Anulación/override)

## El polimorfismo del tiempo de compilación

El polimorfismo de tiempo de compilación es un método y los operadores están sobrecargados. En el método de sobrecarga de método realiza la tarea diferente en los diferentes parámetros de entrada.

## Polimorfismo en tiempo de ejecución

El polimorfismo en tiempo de ejecución se realiza mediante herencia y funciones virtuales. La anulación del método se llama polimorfismo en tiempo de ejecución. Al anular un método, cambia el comportamiento del método para la clase derivada. Sobrecargar un método implica simplemente tener otro método con el mismo prototipo.

**Nota**: No confundir la sobrecarga de métodos con la invalidación de métodos, son conceptos diferentes, no relacionados. La sobrecarga de métodos no tiene nada que ver con la herencia o los métodos virtuales.

Los siguientes son ejemplos de métodos tienen diferentes sobrecargas:

```
void area(int lado);
void area(int l, int b);

void area(float radio);
```

Ejemplo de Sobrecarga de métodos:

```
using System;

public class EjemploSobrecarga
{
    class Program
    {
        public class Imprimir
        {

            public void mostrar(string nombre)
            {
                Console.WriteLine("Tu nombre es: " + nombre);
            }

            public void mostrar(int edad, float notas)
```

```
            {
                Console.WriteLine("Tu edad es: " + edad);
                Console.WriteLine("Tu nota es:" + notas);
            }
        }

        static void Main(string[] args)
        {

            Imprimir objeto = new Imprimir();
            objeto.mostrar("Juan");
            objeto.mostrar(25, 99.50f);
            Console.ReadLine();
        }
    }

}
```

**Nota**: En el código se observa que el método de mostrar se llama dos veces. El método de mostrar funcionará de acuerdo con el número de parámetros y el tipo de parámetros.

Podemos usar la sobrecarga de métodos en una situación en la que necesitemos que una clase pueda hacer algo, pero existe más de una posibilidad de que la información que se proporciona al método que lleva a cabo la tarea sea diferente, es decir, si, por alguna razón, necesitamos un par de métodos que tomen diferentes parámetros, pero que conceptualmente hagan lo mismo.

Método de sobrecarga que muestra muchas formas.

```
using System;

public class ReflectionExample
{
    class Program
    {
        public class Circulo
    {
        public void Area(float radio)
        {
            float a = (float)3.14 * radio;
            // Aquí utilizamos la sobrecarga de funciones con 1 parámetro.
            Console.WriteLine("Área del Círculo: {0}", a);
        }

        public void Area(float l, float b)
        {
            float x = (float)l * b;
            // Aquí utilizamos la sobrecarga de funciones con 2 parámetros.
            Console.WriteLine("Área del Rectángulo: {0}", x);

        }

        public void Area(float a, float b, float c)
        {
            float s = (float)(a * b * c) / 2;
            // Aquí utilizamos la sobrecarga de funciones con 3 parámetros.
            Console.WriteLine("Área del Círculo: {0}", s);
```

```
        }
    }

    static void Main(string[] args)
    {
        Circulo objeto = new Circulo();
        objeto.Area(3.0f);
        objeto.Area(30.0f, 30.0f);
        objeto.Area(3.0f, 5.0f, 8.0f);
        Console.ReadKey();
    }
}


}
```

Si utiliza la sobrecarga para el método, existen algunas restricciones que impone el compilador. La regla es que las sobrecargas deben ser diferentes en su firma, lo que significa el nombre y el número y tipo de parámetros.

No hay límite a la cantidad de sobrecargas que puede tener de un método. Simplemente se declaran en una clase, como si fueran métodos diferentes que tuvieran el mismo nombre.

## Anulación (Overriding) del método

Anular significa cambiar la funcionalidad de un método sin cambiar la firma. Podemos anular una función en la clase base creando una función similar en la clase derivada. Esto se hace mediante el uso de palabras clave virtual/override.

El método de clase base tiene que estar marcado con una palabra clave virtual y podemos anularlo en una clase derivada usando una palabra clave override.

El método de la clase derivada anulará completamente el método de la clase base, es decir, cuando hacemos referencia al objeto de la clase base creada al convertir el objeto de la clase derivada, se llamará a un método en la clase derivada.

Ejemplo:

```
using System;

public class EjemploOverriding
{
    // Clase Base
    public class ClaseBase
    {
        public virtual void Metodo1()
        {
            Console.Write("Método Clase Base");
        }
    }
```

```
    // Clase Derivada
    public class ClaseDerivada : ClaseBase
    {
        public override void Metodo1()
        {
            Console.Write("Método Clase Derivada");
        }
    }
    // Usando clase base y clase derivada
    public class Prueba
    {
        public void PruebaMetodo()
        {
            // llamando al método anulado
            ClaseDerivada objetoDerivada = new ClaseDerivada();
            objetoDerivada.Metodo1();
            // llamando al método de la clase base
            ClaseBase objetoBase = new ClaseBase();
            objetoBase.Metodo1();
        }
    }

    static void Main()
    {
        Prueba p = new Prueba();
        p.PruebaMetodo();
        Console.ReadKey();
    }


}
```

**Resultado**:

```
Método Clase DerivadaMétodo Clase Base
```

**FIN**