UNIDAD 7. PROGRAMACIÓN ORIENTADA A OBJETOS DESDE C#- Conceptos Básicos 2

UNIDAD 7. PROGRAMACIÓN ORIENTADA A OBJETOS DESDE C#- Conceptos Básicos

1. Concepto de POO

Paradigma de programación que pretende mejorar aspectos de la programación imperativa tradicional tales como:

- Abstracción con la que representamos el problema.
- Portabilidad del código y por tanto su reusabilidad.
- Modularidad del código y por tanto legibilidad.

Cuando programamos en lenguaje orientado a objetos lo que se debe hacer es atacar los problemas dividiéndolos en unidades lógicas denominadas objetos, que colaborarán entre ellos para resolver el problema. Nos preguntaremos que tipos de objetos intervienen en un problema e intentaremos describirlos de forma abstracta libres de cuestiones de implementación y representación. A esta definición abstracta completa y no ambigua le denominaremos **TAD (Tipo Abstracto de Datos).**
Un objeto es un conjunto de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

2. Definición de clases

Una clase es la implementación total o parcial de un TAD, es decir la definición de las características concretas de un determinado tipo de objeto (cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo). Por esta razón, se suele decir que el tipo de dato de un objeto es la clase que define las características del mismo.

Podremos definir las clases con 2 naturalezas:

- Como módulo: Organización y encapsulación de software.
- Como Tipo: Define un TAD con sus atributos y operaciones.

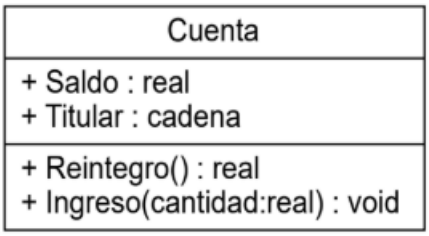
Una Clase está formada por los siguientes elementos:

• Un nombre: que describe a la clase.

• Atributos: Son datos necesarios para describir los objetos (instancias) creados a partir de la clase. La combinación de sus valores determina el estado de un objeto.

• Roles: relaciones que una clase establece con otras clases.

• Operaciones, Métodos, Servicios. Describen las operaciones posibles sobré un objeto de esa clase, además deberían ser el único modo de acceder a los atributos.



- Para **definir** una clase seguiremos la siguiente sintaxis

```
class <nombreClase>
{
      <miembros>
}
```

De este modo se definiría una clase de nombre **<nombreClase>** que describe a la clase y cuyos miembros son los definidos **en <miembros>.** Los miembros de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma. Por ahora vamos a considerar que los miembros únicamente pueden ser atributos o métodos.

- Un campo o atributo es un dato común a todos los objetos de una determinada clase.

```
<tipoCampo> <nombreCampo>;
```

En **<tipoCampo>** hemos de indicar cuál es el tipo de dato del objeto que vamos a crear, puede ser de los predefinidos en la BCL u otro que nosotros hallamos definido.

- Un método es un conjunto de instrucciones a las que se les asocia un nombre. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido.

```
<tipoDevuelto> <nombreMétodo> (<parametros>)
{
      <instrucciones>
}
```

Todo método puede devolver o no un objeto como resultado de la ejecución de las instrucciones que lo forman esto lo hará a través de **<tipoDevuelto>.** También puede recibir en cada llamada una lista de objetos a los que podrá acceder durante la ejecución de sus instrucciones, esta lista se refiere a los **<parametros>.**

Ejemplo de definición de una clase de nombre Persona que dispone de tres campos y método de nombre Cumpleaños:

```
class Persona
  {
      string Nombre; //Campo de cada objeto Persona que almacena el nombre
      int Edad;    // Campo de cada objeto Persona que almacena su edad
      string NIF;       // Campo de cada objeto Persona que almacena su NIF

      void Cumpleaños()  // Incrementa en uno de la edad del objeto Persona
      {
          Edad++;
      }
  }
```

Según esta definición, todos los objetos de clase Persona incorporarán campos que almacenarán cuál es el nombre de la persona que cada objeto representa, cuál es su edad y cuál es su NIF.

Pero **¿Qué Es Un Objeto?** De manera informal podemos decir que es una instancia en memoria de una clase creada en tiempo de ejecución.

En una misma clase pueden definirse varios métodos con el mismo nombre siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como **sobrecarga de métodos**, y es posible ya que el compilador sabrá a cual llamar a partir de los **<parámetros>** especificados. Sin embargo, lo que no se permite es definir varios métodos que únicamente se diferencien en su valor de retorno, porque en este caso no se identificarían en la llamada.

3. Creación de objetos

Para crear objetos de una determinada clase se utiliza el operador new, cuya sintaxis es:

new <nombreTipo>(<parametros>)

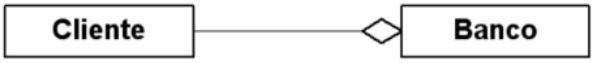
Este operador crea un nuevo objeto del tipo cuyo nombre se le indica. Para ello llama al constructor del objeto mas apropiado según los valores que se le pasen en <parametros>, y devuelve una referencia a la dirección de memoria dinámica donde se ha creado el objeto.
La sintaxis usada para llamar a los métodos y a los campos de un determinado objeto es la siguiente:

<objeto>.<campo>
<objeto>.<método>(<parámetros>)

Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y cambiar su valor por 20 se haría y si quisieramos incrementar la edad en 1 llamariamos al método Cumpleaños.

```
p.Edad = 20;
p.Cumpleaños();   // El método no toma parámetros, luego
                  //no le pasamos ninguno
```

Entre las clases nos podemos encontrar distintos modos de relación, con respecto a sus miembros instanciados de otra clase. Esta relación se llama de Composición (subojetos) o Agragación (referencia)
La **agregación** indica independencia de los objetos, esto es si desaparece el contenedor no desaparece el agregado. Ej. Si desaparece el banco no desaparece el cliente.



La **composición** indica dependencia de los objetos, esto es si desaparece el contenedor, también desaparece el agregado. Ej: Si desaparece el banco desaparecen sus sucursales.

3.1. Constructores

El constructor de un objeto no es más que un método definido en la definición de su tipo que tiene el mismo nombre que la clase a la que pertenece el objeto y no tiene valor de retorno, esto es así porque como siempre devuelve una referencia al objeto no tiene sentido incluir el campo <valor de retorno>.
El constructor recibe ese nombre debido a que su código suele usarse precisamente para construir el objeto, es decir para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona
{
        string Nombre;
        int Edad;
        string NIF;

        public void Cumpleaños()
        {
            Edad++;
        }
        //Constructor
        public Persona(string nombre, int edad, string nif)
        {
            Nombre = nombre;
            Edad = edad;
            NIF = nif;
        }
}
```

Como se ve en el código, el constructor toma como parámetros los valores con los que deseemos inicializar el objeto a crear. Gracias a él, podemos crear un objeto Persona de nombre José, de 22 años de edad y NIF 12344321-A así:

```
new Persona("José", 22, "12344321-A")
```

Operador new → todo tipo creado mediante class, será un tipo referencia por lo que siempre se deberá utilizar el operador new para crear el objeto mediante cualquiera de los constructores implementados en la clase.

```
Persona persona=new Persona("José", 22, "12344321-A")
```

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por new en una variable del tipo apropiado para el objeto creado.
Ejemplo:

```
Persona p;    // Creamos variable p
//Almacenamos en p el objeto creado con new
p = new Persona("Jose", 22, "12344321-A");
//Forma compacta de hacer lo anterior
Persona p = new Persona("José", 22, "12344321-A");
```

A partir de este momento la variable p representa a una persona llamada José de 22 años y NIF 12344321-A.

En realidad un objeto puede tener múltiples constructores, aunque para diferenciar a unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan.

3.2. Constructor por defecto

No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones. Es decir, como si se hubiese definido de esta forma:

```
<nombreTipo>(){}
```

Debido a esto será posible crear un objeto de un tipo al que no le hayamos creado ningún constructor. Pero hay que tener en cuenta una cosa: el constructor por defecto es sólo incluido por el compilador si no hemos definido ningún otro constructor. Por tanto, si tenemos una clase en la que hayamos definido algún constructor con parámetros pero ninguno sin parámetros no será válido crear objetos de la misma llamando al constructor sin parámetros, pues el compilador no lo habrá definido automáticamente.

3.3. Constructor copia

En una clase se puede definir el constructor copia, que como su propio nombre indica, se implementaría para realizar una copia de un objeto de esa clase en otro. La signatura de un constructor copia sería la misma que en el constructor por defecto pero pasándole por parámetro un objeto tipo que la clase.

```
public Persona(Persona persona)
{
    Nombre = persona.nombre;
    Edad = persona.edad;
    NIF = persona.nif;
}
```

```
Persona persona=new Persona("José", 22, "12344321-A");
Persona persona2=new Persona(persona);
```

Por lo que, para hacer una copia de un objeto, siempre tendremos que construir una instancia nueva como en el anterior ejemplo. Y nunca una asignación directa, puesto que estaríamos referenciando al mismo objeto en memoria.

```
Persona persona2=persona;  //Error, nunca hacer esto para
                           //copiar un objeto
```

3.4. Destructores

El destructor gestiona cómo se destruyen los objetos de un determinado tipo de dato. Este método suele ser útil para liberar recursos tales como los ficheros o las conexiones de redes abiertas que el objeto a destruir estuviese acaparando en el momento en que se fuese a destruir.
La destrucción de un objeto es realizada por el recolector de basura cuando realiza una recolección de basura y detecta que no existen referencias a ese objeto ni en pila, ni en registros ni desde otros objetos sí referenciados. Las recolecciones se inician automáticamente cuando el recolector detecta que queda poca memoria libre o que se va a finalizar la ejecución de la aplicación, aunque también puede forzarse llamando al método Collect() de la clase System.GC. La sintaxis que se usa para definir un destructor es la siguiente:

```
~<nombreTipo>()
{
      <código>
}
```

Tras la ejecución del destructor de un objeto de un determinado tipo siempre se llama al destructor de su tipo padre, formándose así una cadena de llamadas a destructores que acaba al llegarse al destructor de object. Éste último destructor no contiene código alguno, y dado que object no tiene padre, tampoco llama a ningún otro destructor.

Los destructores no se heredan. Sin embargo, para asegurar que la cadena de llamadas a destructores funcione correctamente si no incluimos ninguna definición de destructor en un tipo, el compilador introducirá en esos casos una por nosotros de la siguiente forma:

```
~<nombreTipo>() {}
```

El recolector de basura siempre tiene acceso a los objetos aunque no se hayan creado instancias a los mimos y no sean accesibles para el programador.

Es importante recalcar que no es válido incluir ningún modificador en la definición de un destructor, ni siquiera modificadores de acceso, ya que como nunca se le puede llamar explícitamente no tiene ningún nivel de acceso para el programador.

3.5. Operador this. Referencia al objeto actual

Dentro del código de cualquier método de un objeto siempre es posible hacer referencia al propio objeto usando la palabra reservada this. Por ejemplo, el constructor de la clase Persona escrito anteriormente se puede rescribir así usando this:

```
public Persona(string Nombre, int Edad, string NIF)
{
            this.Nombre = Nombre;
            this.Edad = Edad;
            this.NIF = NIF
}
```

Es decir, dentro de un método con parámetros cuyos nombres coincidan con campos, se da preferencia a los parámetros y para hacer referencia a los campos hay que prefijarlos con this.

Un uso más frecuente de this en C# es el de permitir realizar llamadas a un método de un objeto desde código ubicado en métodos del mismo objeto.

Finalmente, una tercera utilidad de this es permitir escribir métodos que puedan devolver como objeto el propio objeto sobre el que el método es aplicado. Para ello bastaría usar una instrucción return this; al indicar el objeto a devolver.

Ejemplo del programa de Persona:

```
class Persona
{
    string Nombre;
    int Edad;
    string NIF;


     public Persona(string nombre, int edad, string nif)
    {
        Nombre = nombre;
        Edad = edad;
        NIF = nif;
    }
    public void Cumpleaños()
    {
        Edad++;
    }
    //Método que muestra la edad de la persona cuando pasen X años
    //utilizamos el operador this para llamar al método Cumpleaños
    public void MostrarEdadFutura(int años)
    {
        for (int i = 0; i < años; i++) this.Cumpleaños();
        Console.WriteLine(Edad);
    }
}
class Program
{
```

```
        static void Main()
        {
            Persona p = new Persona("Pepe", 10, "22112211L");
            p.MostrarEdadFutura(5);
        }
    }
```

Ejemplo de programa que desplaza un asterisco por la pantalla x posiciones con un ángulo determinado:

```
  class PuntoConsola
    {
        private ushort Fila;
        private ushort Columna;


        public PuntoConsola(ushort fila, ushort columna) //Constructor
        {
            this.Fila = fila;
            this.Columna = columna;
        }
        public PuntoConsola(PuntoConsola punto)//Constructor copia
        {
            this.Fila = punto.Fila;
            this.Columna = punto.Columna;
        }
        //Muestra un asterisco de un color determinado en un punto
        //Fila, Columna de la pantalla
        public void Muestra(ConsoleColor color)
        {
            ushort x = Columna;
            ushort y = (ushort)(24 - Fila);
            Console.SetCursorPosition(x, y);
            Console.ForegroundColor = color;
            Console.Write("*");
        }
        //Desplaza Fila, Columna el número de posiciones y con el angulo
        //indicado
        public void Desplaza(ushort numPosiciones, double angulo)
        {
            double anguloRad = angulo * Math.PI / 180d;
            Fila += (ushort)(numPosiciones * Math.Sin(anguloRad));
            Columna += (ushort)(numPosiciones * Math.Cos(anguloRad));
        }
    }
    //Clase con el programa principal
    class Program
    {
        static void Main(string[] args)
        {
            //Creación del objeto punto y de otro objeto copia de este
            PuntoConsola punto = new PuntoConsola(12, 40);
            PuntoConsola puntoCopia = new PuntoConsola(punto);
            //Llamada a los métodos para desplazar y mostrar los puntos
            punto.Muestra(ConsoleColor.DarkBlue);
            Console.ReadKey(true);
            punto.Desplaza(4, 45d);
            punto.Muestra(ConsoleColor.Blue);
```

```
            Console.ReadKey(true);
            punto.Desplaza(10, 180d);
            puntoCopia.Desplaza(15, 0d);
            punto.Muestra(ConsoleColor.DarkCyan);
            Console.ReadKey(true);
            puntoCopia.Muestra(ConsoleColor.DarkRed);
            Console.ReadKey(true);
            punto.Desplaza(10, 120d);
            puntoCopia.Desplaza(10, 160d);
            punto.Muestra(ConsoleColor.Cyan);
            Console.ReadKey(true);
            puntoCopia.Muestra(ConsoleColor.Red);
            Console.ReadKey(true);

            Console.SetCursorPosition(20, 20);
        }
    }
```

3.6. Miembros de tipo o clase

En realidad, dentro la definición de un tipo de dato no tienen porqué incluirse sólo definiciones de miembros comunes a todos sus objetos, sino también pueden definirse miembros ligados al tipo como tal y no a los objetos del mismo. Para ello basta preceder la definición de ese miembro de la palabra reservada static. Para acceder a un miembro de clase ya no es válida la sintaxis hasta ahora vista de <objeto>.<miembro>, pues al no estar estos miembros ligados a ningún objeto no podría ponerse nada en el campo <objeto>. La sintaxis a usar para acceder a estos miembros será <nombreClase>.<miembro>.

```
class A
{
   int x;
   static int y;
   static void Incrementa()
   {
      x++; //ERROR: x es  miembro de objeto e Incrementa() es de clase
    }
}
```

Los objetos de clase A sólo van a disponer del campo x, mientras que el campo y va a pertenecer a la clase A. Por esta razón se dice que los miembros con modificador static son miembros de tipo y que los no lo tienen son miembros de objeto.

Es importante matizar que si definimos una función como static, entonces el código de la misma sólo podrá acceder implícitamente a otros miembros static del tipo de dato al que pertenezca. O sea, no se podrá acceder a ni a los miembros de objeto del tipo en que esté definido ni se podrá usar this ya que el método no está asociado a ningún objeto.

También hay que señalar que los métodos estáticos no entran dentro del mecanismo de redefiniciones. Dicho mecanismo sólo es aplicable a métodos de objetos, que son de quienes puede declararse variables y por tanto puede actuar el polimorfismo. Por ello, incluir los modificadores: virtual, override o abstract al definir un método static es considerado erróneo por el compilador.

Eso no significa que los miembros static no se hereden, sino tan sólo que no tiene sentido redefinirlos.

3.7. Definiendo Accesores y Mutadores

Los accesores y mutadores son un elemento esencial para la programación, ya que nos permite acceder desde otras clases, a atributos que en principios son privados, pero que el programador decide hacer público para el acceso o para la mutación a través de estos métodos
Para el accesor, crearemos un método con el prefijo get seguido del nombre del atributo.
Para el mutador, crearemos un método con el prefijo set seguido del nombre del atributo.
Lo podemos mostrar en el siguiente ejemplo:

```
public ushort GetFila()
{
   return Fila;
}
public void SetFila(ushort columna)
{
     Columna=columna;
}
```

Resumen Directrices Generales Iniciales De Implementación

- Definir datos miembro o atributos privados (Encapsulación).
- Definir constructores, y destructor. Este último, solo si es necesarios.
- Definir accesores/mutadores o propiedades solo cuando sea necesario.
- Definir operaciones sobre el objeto sobre el objeto como métodos públicos.
- Recordar mantener siempre la integridad del estado de nuestro objeto.

4. Herencia

El mecanismo de herencia es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera, a la que se le suele llamar clase hija, será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda, a la que se le suele llamar clase padre o clase base. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <nombreHija>:<nombrePadre>
{
      <miembrosHija>
}
```

A los miembros definidos en <miembrosHijas> se le añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase Persona puede crearse una clase Trabajador así:

```
class Trabajador : Persona
{
  public int Sueldo;

  public Trabajador(string nombre, int edad, string nif, int sueldo):
  base(nombre, edad, nif)
  {
      Sueldo = sueldo;
  }
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que cada trabajador gane. Nótese además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

: base(<parametrosBase>)

A esta estructura se le llama inicializador base y no es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto **:base(),** por ello hay que estar seguros de que si no se incluye base en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

4.1. La clase primigenia: system.object

Ahora que sabemos lo que es la herencia es el momento apropiado para explicar que en .NET todos los tipos que se definan heredan implícitamente de la clase System.Object predefinida en la BCL, por lo que dispondrán de todos los miembros de ésta. Por esta razón se dice que System.Object es la raíz de la jerarquía de objetos de .NET.

A continuación vamos a explicar cuáles son estos métodos comunes a todos los objetos:

public virtual bool Equals(object o): Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro. Devuelve true si ambos objetos son iguales y false en caso contrario.

public virtual int GetHashCode(): Devuelve un código de dispersión (hash) que representa de forma numérica al objeto sobre el que el método es aplicado. GetHashCode() suele usarse para trabajar con tablas de dispersión, y se cumple que si dos objetos son iguales sus códigos de dispersión serán iguales, mientras que si son distintos la probabilidad de que sean iguales es ínfima.

public System.Type GetType(): Devuelve un objeto de clase System.Type que representa al tipo de dato del objeto sobre el que el método es aplicado. A través de los métodos ofrecidos por este objeto se puede acceder a metadatos sobre el mismo como su nombre, su clase padre, sus miembros, etc.

public virtual string ToString(): Devuelve una representación en forma de cadena del objeto sobre el que el método es aplicado, lo que es muy útil para depurar aplicaciones ya que permite mostrar con facilidad el estado de los objetos.

Veamos un ejemplo al que hemos añadido el siguiente método en la clase Trabajador:

```
public override bool Equals(object obj)
{
   if (obj is Trabajador)
   if (this.Sueldo==((Trabajador)obj).Sueldo) return true;
   return false;
}
```

Modificar la Main para poder probar el Nuevo método

```
class Program
{
  static void Main(string[] args)
  {
      Trabjador luis, maria;
      luis = new Trabajador("Luis", 22,"77588261Z");
      maria = new Trabajador("Maria", 27,"74567421A", 1200);
      luis.Sueldo=maria.Sueldo;
      if(luis.Equals(maria))    Console.WriteLine("Los    trabajadores
tienen el mismo sueldo");
      else  Console.WriteLine("Los  trabajadores  no  tienen  el  mismo
sueldo");
}
```

4.2. Métodos Virtuales

Cuando en la definición de un método de la clase padre va precedido con la palabra reservada virtual es posible cambiar dicha definición en la clase hija. A este tipo de métodos se les llama métodos virtuales, y la sintaxis que se usa para definirlos es la siguiente:

```
virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)
 {
       <código>
 }
```

Si en alguna clase hija quisiésemos dar una nueva definición del <código> del método, simplemente lo volveríamos a definir en la misma pero sustituyendo en su definición la palabra reservada virtual por override. Es decir, usaríamos esta sintaxis:

```
override <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
      <nuevoCódigo>
}
```

Cuando definamos un método como override ha de cumplirse que en alguna clase antecesora (su clase padre, su clase abuela, etc.) de la clase en la que se ha realizado la definición del mismo exista un método virtual con el mismo nombre que el redefinido. Esta técnica se llama **sobreescritura o refinamiento de métodos.**

Ejemplo de redefinición del método virtual Cumpleaños de la clase trabajador:

```
        class Persona
      {
         public string Nombre;
         public int Edad;
         public string NIF;
         public virtual void Cumpleaños()
         {
                 Edad++;
                Console.WriteLine("Has cumplido un año más");
         }
         public Persona (string nombre, int edad, string nif)
         {
              Nombre = nombre;
              Edad = edad;
          NIF = nif;
         }
      }
      class Trabajador: Persona
      {
          public int Sueldo;
          public Trabajador(string nombre, int edad, string nif, int sueldo)
         :base(nombre, edad, nif)
         {
            Sueldo = sueldo;
         }
         public Trabajador(string nombre, int edad, string nif)
         :base(nombre, edad, nif)
         {
            Sueldo = 0;
         }

         public override void Cumpleaños()
         {
      Edad++;
            Console.WriteLine("Feliz cumpleaños te desea la empresa");
         }

         public static void Main()
         {
             Persona p = new Persona("Carlos", 22,"77588261Z");
             Trabajador t = new Trabajador("Ana",22,"7788260Z", 2000);
            t.Cumpleaños();
            p.Cumpleaños();
         }
}
```

Toda redefinición de un método virtual ha de mantener los mismos modificadores de acceso que el método original, por ese motivo también se ha mantenido en la redefinición de Cumpleaños() el modificador public.

5. Polimorfismo

El polimorfismo es otro de los pilares fundamentales de la programación orientada a objetos. Bajo el nombre de POLIMORFISMO distinguiremos en la POO:

- Polimorfismo funcional o sobrecarga.
- Polimorfismo de datos o inclusión.
- Polimorfismo paramétrico o tipos genéricos (se verá en temas posteriores)

5.1 Polimorfismo funcional o sobrecarga

Como ya vimos en el tema anterior de métodos, la sobrecarga es la capacidad de tener dentro de una misma clase más de un método con el mismo nombre, pero ojo tienen que tener diferente signatura.

En C# dos métodos tienen diferente signatura si:

- Tienen diferente tipo de retorno.
- Tienen diferente número de parámetros.
- Teniendo el mismo número de parámetros algún tipo es diferente.
- Teniendo el mismo número de parámetros y el mismo tipo alguno tiene el modificador ref o out.

5.2 Polimorfismo de datos o de inclusión

Es la capacidad de un identificador de hacer referencia a instancias de distintas clases durante su ejecución. **Esto se logra a través del principio de sustituición de Liskov también conocido como UPCASTING.**

Se puede decir que ocurre cuando se almacenan objetos de una determinada subclase en un identificador del tipo de la superclase a la que pertenecen.

Sólo se podrá acceder a través de dicho identificador a los miembros comunes a ambas clases.

```
Persona p = new Trabajador("Ana",22,"7788260Z", 2000);
```

La palabra reservada base que incluye C# tiene la utilidad ya conocida de la reutilización del constructor de una clase padre y una nueva, que permite devolver una referencia al objeto actual semejante a this pero con la peculiaridad de que los accesos a ella son tratados como si el verdadero tipo fuese el de su clase base.

En el siguiente ejemplo podemos ver como el identificador transporte que es de la superclase Vehículo durante la ejecución puede referenciar a dos clases diferente Camión y Coche, pero sólo podemos llamar a métodos definidos en Vehículo y no a ninguno específico de Camión o Coche si los hubiere, también vemos la utilización de **base**.

```
class Vehiculo
{
        private string matricula;
        private ushort cilindradaCC;
        private string color;
        private string marca;
        private ushort numPlazas;
```

```
    public Vehiculo(string matricula,ushort cilindradaCC,
                    string color,string marca,ushort numPlazas)
    {
        this.matricula = matricula;
        this.cilindradaCC = cilindradaCC;
        this.color = color;
        this.marca = marca;
        this.numPlazas = numPlazas;
    }

    virtual public void VerDatos()
    {
        Console.Write("Matricula: " ,matricula);
        Console.Write("cilindrada: {0} cc\n", cilindradaCC);
        Console.Write("Color: {0}\n", color);
        Console.Write("Marca: {0}\n", marca);
        Console.Write("Plazas: {0}\n", numPlazas);
    }
```

```
    virtual public void LeeDatos()
    {
        Console.Write("Introduzca la Matricula: ");
        matricula = Console.ReadLine();
        Console.Write("Introduzca la cilindrada: ");
        cilindradaCC = ushort.Parse(Console.ReadLine());
        Console.Write("Introduzca la Color: ");
        color = Console.ReadLine();
        Console.Write("Introduzca la Marca: ");
        marca = Console.ReadLine();
        Console.Write("Introduzca la Plazas: ");
        numPlazas = ushort.Parse(Console.ReadLine());
    }
}

class Coche : Vehiculo
{
    public enum Categoria { Berlina, Coupe, Sedan, Cabrio,
                            TodoTerreno, MonoVolumen };
    private Categoria categoria;

    public Coche(string matricula,ushort cilindradaCC,
                 string color,string marca,ushort numPlazas,
                 Categoria categoria)
    :base(matricula, cilindradaCC, color, marca, numPlazas)
    {
        this.categoria = categoria;
    }

    override public void VerDatos()
    {
        base.VerDatos();
        Console.Write("Categoria: {0}\n", categoria);
```

```
        }

        override public void LeeDatos()
        {
            base.LeeDatos();
            categoria = (Categoria) Enum.Parse(typeof(Categoria),
                                                Console.ReadLine());
        }
    }
    class Camion : Vehiculo
    {
        private ushort numEjes;
        private ushort cargaMaximaKg;
        public Camion(string matricula,ushort cilindradaCC,
                      string color, string marca,ushort numPlazas,
                      ushort numEjes,ushort cargaMaximaKg)
              :base (matricula, cilindradaCC, color, marca, numPlazas)
        {
          this.numEjes = numEjes;
          this.cargaMaximaKg = cargaMaximaKg;
        }
```

```
        override public void VerDatos()
        {
            base.VerDatos();
            Console.Write("Número de ejes: {0}\n", numEjes);
            Console.Write("Carga Máxima: {0} Kg\n", cargaMaximaKg);
        }

        override public void LeeDatos()
        {
            base.LeeDatos();
            Console.Write("Introduzca el número de ejes: ");
            numEjes = ushort.Parse(Console.ReadLine());
            Console.Write("Introduzca la carga máxima: ");
            numEjes = ushort.Parse(Console.ReadLine());
        }
        public void Mensage()
       {Console.WriteLine("ESTO ES UN COCHE"); }
    }
    class program
    {
        static void Main()
        {
            Vehiculo transporte;
            Console.WriteLine("\nDATOS DE CAMION");
            transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN",
                                    5, 5,10000);
            transporte.VerDatos();
            Console.WriteLine("\nDATOS DE COCHE");
            transporte = new Coche("1021-DRG", 1600, "Azul Celeste",
                                   "SEAT", 5, Coche.Categoria.Sedan);
             transporte.VerDatos();
```

```
            //transporte.Mensage();//Esta llamada
                                   //daría error, porque este método
                                   //es propio de la subclase coche
        }
    }
```

Al ejecutarse transporte.VerDatos() al ser VerDatos() un método virtual en Vehículo, cuando transporte reciba el mensaje, antes de ejecutarse el cuerpo del método definido en vehículo, comprobará a que objeto referencia realmente transporte y si es a una subclase de Vehículo verá si tiene sobrescrito (override) el método VerDatos() y si es así será el que ejecute, en caso contrario ejecutará el cuerpo definido en Vehículo. Esto hace que hasta el momento de la ejecución sabremos que VerDatos(), "Se **ligará dinámicamente** (concepto de ligadura dinámica)" para ejecutarse.

Podremos hacer la operación contraria, llamada **DOWNCASTING,** siempre y cuando se le asigne a la variable de tipo hijo una referencia del padre que a su vez esté referenciada sobre una del hijo.

```
Vehiculo transporte=new Camion("1020-DRG",5000,"Blanco","MAN",5,5, 10000);
Camion c = (Camion)transporte;
```

5.3 Operadores de utilidad para el principio de sustituición de Liskov

• Operador **is** → sirve para preguntar a un objeto si es de un determinado tipo.

```
Vehiculo transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN", 5, 5, 10000);
if (transporte is Camion) { Camion c = (Camion)transporte; }
```

• Operador **as** → (Es una sintaxis sugar de is), realiza directamente el downcasting y si no puede, asigna null.

```
Vehiculo transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN", 5, 5, 10000);
Camion c = transporte as Camion;
```

Equivaldría a hacer:

```
Camion c = transporte is Camion ? (Camion)transporte : null;
```

• Operador de coalescencia nulo **??** → (Es una sintaxis sugar de algunos casos del operador ?:). Sirve para simplificar expresiones del tipo:

```
Vehiculo transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN", 5, 5, 10000);
Vehiculo transporte = camion != null ? camion : new Camion();
```

Y en su lugar poner…

```
Vehiculo transporte = camion??new Camion();
```

Se podría usar en el downcasting de la siguiente forma...

```
Vehiculo transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN", 5, 5, 10000);
Camion c = transporte as Camion ?? new Camion();
```

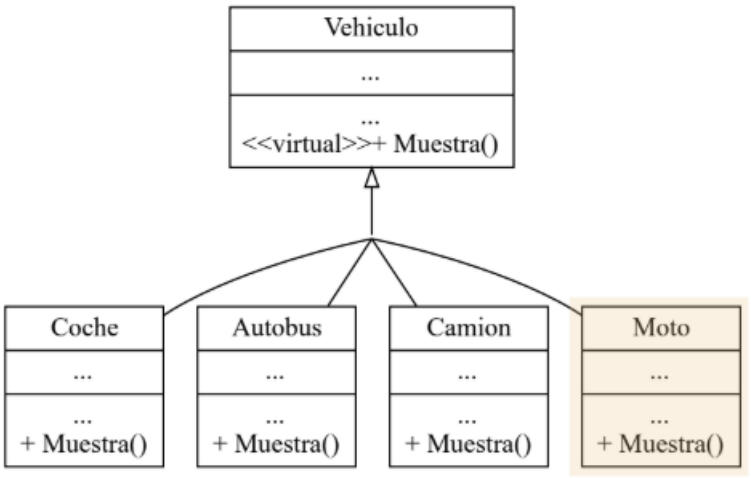
- Cláusula case de switch para objetos (con when opcional):

```
Vehiculo transporte = new Camion("1020-DRG", 5000, "Blanco", "MAN", 5, 5, 10000);

        switch (transporte)
        {
            case null:
                break;
            case Camion x when x.GetNumEjes() == 3:
                break;
            case Camion x:
                Console.WriteLine(x.GetNumEjes());
                break;
        }
```

Entra en case si se puede hacer el downcasting al nuevo identificador (x en este caso), el ámbito de existencia del nuevo identificador será hasta el siguiente case.
Si añadimos una especificación con when tiene que estar antes de la generalización.

¿Para Que Sirve Este Polimorfismo De Datos?
En ocasiones el software cambia y se añaden nuevas especificaciones. Con el polimorfismo de datos, podremos adaptarnos a futuros cambios (Nuevas formas de un objeto), sin realizar cambios traumáticos y costosos en las implementaciones anteriores.



Supongamos que añadimos las Motos como nuevo tipo de Vehiculo,
Podremos hacer...
p.Entra(new Moto());
Sin tener que cambiar nada en la clase Parking.

```
class Parking
{
    public void Entra(Vehiculo v)
    {
        Console.WriteLine("Estas añadiendo un: ");
        // Se enlazará dinámicamente
        // con el Mostrar() de la subclase subyacente.
        v.VerDatos();
    }
}
```

```
class Programa
{
    static void Main()
    {
        Parking p = new Parking();
        // En cada llamada a Entra()
        // la subclase se sustituye por la superclase.

        p.Entra(new Coche());
        p.Entra(new Camion());
        p.Entra(new Moto());

    }
}
```

6. Encapsulación

Ya hemos visto que la herencia y el polimorfismo eran dos de los pilares fundamentales en los que se apoya la programación orientada a objetos. Pues bien, el tercero y último es la encapsulación, que es un mecanismo que permite a los diseñadores de tipos de datos determinar qué miembros de los tipos pueden ser utilizados por otros programadores y cuáles no. Las principales ventajas que ello aporta son:
Facilitar a los programadores que vaya a usar el tipo de dato el aprendizaje de cómo trabajar con él, pues se le pueden ocultar todos los detalles relativos a su implementación interna.
Facilitar al creador del tipo la posterior modificación del mismo, pues si los programadores clientes no pueden acceder a los miembros no visibles, sus aplicaciones no se verán afectadas si éstos cambian o se eliminan.
La encapsulación se consigue añadiendo modificadores de acceso en las definiciones de miembros y tipos de datos. Estos modificadores son partículas que se les colocan delante para indicar desde qué códigos puede accederse a ellos. Por defecto se considera que los miembros de un tipo de dato sólo son accesibles desde código situado dentro de la definición del mismo.

Los modificadores de miembro son los siguientes:

- private: Sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto.
- public: Puede ser accedido desde cualquier código.
- internal: Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido.
- protected internal: Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido o desde clases que deriven de la clase donde se ha definido.
- protected: Desde una clase sólo puede accederse a miembros protected de objetos de esa misma clase o de subclases suyas.

Ejemplo de acceso a miembros protegidos:

```
public class A
{
   protected int x;
   static void F(A a, B b, C c)
   {
      a.x = 1;  // Ok
      b.x = 1;  // Ok
      c.x = 1;  // OK
   }
}
public class B : A
{
   static void F(A a, B b, C c)
   {
      //a.x = 1;  // Ha de accederse a traves de objetos tipo B o C
      b.x = 1;  // Ok
      c.x = 1;  // Ok
   }
}
public class C : B
{
   static void F(A a, B b, C c)
   {
     //a.x = 1; // Error, ha de accederse a traves de objetos tipo C
     //b.x = 1; // Error, ha de accederse a traves de objetos tipo C
       c.x = 1; // Ok
    }
}
```

A lo que no se podrá acceder desde una clase hija es a los miembros protegidos de otros objetos de su clase padre, sino sólo a los heredados, es decir:

```
class A
{
    protected int x=5;
}
class B:A
{
      B(A objeto)
      {
            Console.WriteLine("Heredado x={0} de clase A", x);
            Console.WriteLine(objeto.x); // Error, no es el x heredado
      }
}
```

Si se duda sobre el modificador de visibilidad a poner a un miembro, es mejor ponerle inicialmente el que proporcione menos permisos de accesos, ya que si luego detecta que necesita darle más permisos siempre podrá cambiárselo por otro menos restringido.

Es importante recordar que toda redefinición de un método virtual o abstracto ha de realizarse manteniendo los mismos modificadores que tuviese el método original.

Los modificadores de tipos de datos son:

- internal: Sólo es posible acceder a la clase desde el ensamblado donde se declaró. Es lo considerado por defecto.
- public: Es posible acceder a la clase desde cualquier ensamblado.

También pueden definirse tipos dentro de otros (tipos internos). En ese caso serán considerados miembros del tipo contenedor dentro de la que se hayan definido, por lo que les serán aplicables todos los modificadores válidos para miembros y por defecto se considerará que, como con cualquier miembro, son privados. Para acceder a estos tipos desde código externo a su tipo contenedor además de necesitar los permisos de acceso necesarios, según el modificador de accesibilidad al definirlos, hay que usar la notación <nombreTipoContendor>.<nombreTipoInterno>

Como muestra en este ejemplo:

```
class A  // No lleva modificador, luego se considera que es internal
{
  public class AInterna {}//Si ahora no se pusiese public sería
private
}

class B:A.AInterna //  B deriva de la clase interna AInterna definida
{}                 //dentro de A. Es válido porque A.AInterna es
pública
```

7. Clases abstractas

Una clase abstracta es aquella que forzosamente se ha de derivar si se desea poder crear objetos de la misma o acceder a sus miembros estáticos. Para definir una clase abstracta se antepone abstract a su definición.

La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación sino que se deje en manos de sus clases hijas darla. No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador abstract y sustituyendo su código por un punto y coma (;),

```
    public abstract class A //clase abstracta
    {
        public abstract void F();//Método abstracto
        //no tiene implementación
    }
    //B también ha de definirse como abstracta porque
```

```
//tampoco implementa el método F() que hereda de A
abstract public class B : A
{
    public void G() { }
}
class C : B
{
    public override void F()
   { }
}
```

Como un método abstracto no tiene código no es posible llamarlo. Todo método definido como abstracto es implícitamente virtual, pues si no sería imposible redefinirlo para darle una implementación en las clases hijas de la clase abstracta donde esté definido. Por ello es necesario incluir el modificador override a la hora de darle implementación y es redundante marcar un método como abstract y virtual a la vez (de hecho, hacerlo provoca un error al compilar).

Es posible marcar un método como abstract y override a la vez, lo que convertiría al método en abstracto para sus clases hijas y forzaría a que éstas lo tuviesen que reimplementar si no se quisiese que fuesen clases abstractas.

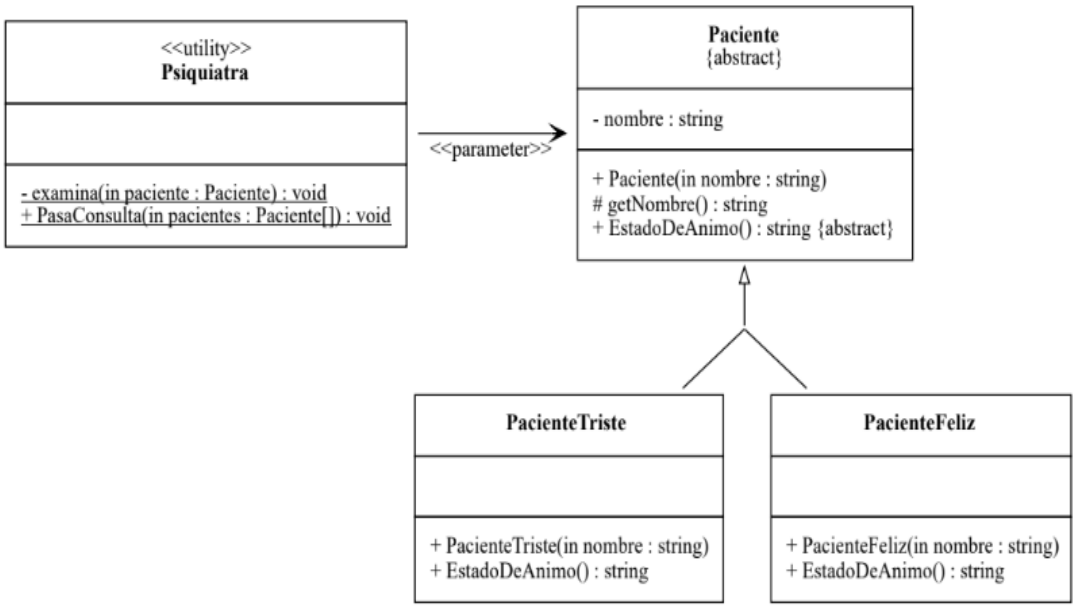
A la hora de redefinir métodos abstractos hay que tener cuidado con una cosa: desde el método redefinidor no es posible usar base para hacer referencia a métodos abstractos de la clase padre, aunque sí para hacer referencia a los no abstractos.

Por ejemplo:

```
abstract class A
{
   public abstract void F();
   public void G(){ }
}
class B : A
{
   public override void F()
   {
      base.G();   // Correcto
      base.F();     // Error, base.F() es abstracto
   }
}
```

A las clases abstractas con todos sus métodos abstractos, se les denomina **Clases Abstractas Puras**.

Si buscamos polimorfismo de datos, lo mejor es buscar la abstracción. Esto significa tratar de llevar la implementación hacia las clases hijas, vaciando de funcionalidad a las superclases.
Veamos un ejemplo de como predecir el futuro con el polimorfismo de datos y la abstracción. Vamos a implementar un programa que simule la consulta de un psiquiatra.
¿Qué pasará si creo una nueva clase de paciente?



7.1 Clases y métodos sellados

Una clase sellada es una clase que no puede tener clases hijas, y para definirla basta anteponer el modificador sealed a la definición de una clase normal. Por ejemplo:

```
sealed class ClaseSellada
{
}
```

Hay que tener en cuenta que sellar reduce enormemente la capacidad de reutilización. En realidad la principal causa de la inclusión de estas clases en C# es que permiten asegurar que ciertas clases críticas nunca podrán tener clases hijas y sus variables siempre almacenarán objetos del mismo tipo. Por ejemplo, para simplificar el funcionamiento del CLR y los compiladores se ha optado por hacer que todos los tipos de datos básicos excepto System.Object estén sellados.

Una utilidad de definir una clase como sellada es que permite que las llamadas a sus métodos virtuales heredados se realicen tan eficientemente como si fuesen no virtuales, pues al no poder existir clases hijas que los redefinan no puede haber polimorfismo y no hay que determinar cuál es la versión correcta del método a la que se ha de llamar.

Aparte de para sellar clases, también se puede usar sealed como modificador en la redefinición de un método para conseguir que la nueva versión del mismo que se defina deje de ser virtual y se le puedan aplicar las optimizaciones arriba comentadas. Un ejemplo de esto es el siguiente:

```
class A
{
     public abstract F();
}
```

```
class B:A
{
    public sealed override F() //F() deja de ser redefinible
    {}
}
```

7.2 Ocultación de miembros

Hay ocasiones en las que interese heredar de una clase sin que ello implique que su clase hija herede sus miembros tal cuales sino con ligeras modificaciones. Si queremos que un método de una clase hija sea ligeramente diferente al método de la clase padre (como por ejemplo que uno devuelva un entero y el otro no) hemos de preceder la definición del método ocultador de la palabra reservada new, así indicaremos explícitamente que queremos ocultar ese método heredado:

```
//ocultación explicita
    class Padre
    {
        public void F() { }
    }
    class Hija : Padre
    {
        new public int F()
        { return 1; }//se utiliza new
    }
```

```
//ocultación implicita
    class Padre
    {
        public void F(){ }
    }
    class Hija : Padre
    {
        public int F()
        { return 1; }
     }
```

Como en C# no se admite que en una misma clase hayan dos métodos que sólo se diferencien en sus valores de retorno, puede pensarse que el código anterior producirá un error de compilación. Sin embargo, esto no es así sino que el compilador lo que hará será quedarse únicamente con la versión definida en la clase hija y desechar la heredada de la clase padre. En el caso de omitir new el compilador generará un aviso, que se evitará al añadir esta palabra.

En realidad la ocultación de miembros no implica los miembros ocultados tengan que ser métodos, sino que también pueden ser campos o cualquiera de los demás tipos de miembro que en temas posteriores se verán. Por ejemplo, puede que se desee que un campo X de tipo int esté disponible en la clase hija como si fuese de tipo string.Tampoco implica que los miembros métodos ocultados tengan que diferenciarse de los métodos ocultadores en su tipo de retorno, sino que pueden tener exactamente su mismo tipo de retorno, parámetros y nombre.

Cuando se oculta un método no se cambia su definición en la clase padre sino sólo en la clase hija, por lo que las llamadas al mismo realizadas a través de variables de la clase padre ejecutarán la versión de dicha clase padre y las realizadas mediante variables de la clase hija ejecutarán la versión de la clase hija.

La ocultación puede ser muy útil al usar la herencia para definir versiones especializadas de clases de uso genérico.

Ejemplo sencillo de todo lo que hemos visto hasta aquí:

```
namespace ConsoleApplication1
{
    abstract class Mueble
    {
        private int Ancho;
        private int Largo;

        public Mueble()
        {
            this.Ancho = 0;
            this.Largo = 0;
        }
        public Mueble(int ancho, int largo)
        {
            this.Ancho = ancho;
            this.Largo = largo;
        }
        public int GetAncho()
        {
            return Ancho;
        }
        public int GetLargo()
        {
            return Largo;
        }
        public void SetAncho(int ancho)
        {
            this.Ancho = ancho;
        }
        public void SetLargo(int largo)
        {
            this.Largo = largo;
        }
        //Método virtual
        public virtual int Calculo()
        {

            return this.Largo * this.Ancho/100;
        }
        public void MostrarAgradecimiento()
        {
            Console.WriteLine("Grácias por haber comprado este mueble
                               en nuestro establecimiento");
        }
    }

    class Silla:Mueble
    {
        //atributos
        private int NumPatas;
        private string Color;
        //constructores
```

```
        public Silla():base()
        {
            NumPatas = 0;
            Color = "";
        }
        public Silla(int numPatas, string color, int ancho,
                     int largo):base(ancho,largo)
        {
            this.NumPatas = numPatas;
            this.Color = color;
        }
        //metodos
        public int GetNumPatas()
        {
            return NumPatas;
        }
        public void SetNumPatas(int numPatas)
        {
            this.NumPatas = numPatas;
        }
        public string GetColor()
        {
            return Color;
        }
        public void SetColor(string color)
        {
            this.Color = color;
        }
        //método sobreescrito
        public override int Calculo()
        {
            Console.WriteLine("Calculo del precio de la silla");
            return base.Calculo();
        }
        //ocultación de método
        public new void MostrarAgradecimiento()
        {
            Console.WriteLine("Grácias por haber comprado esta silla
                               en nuestro establecimiento");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Silla ikea;
            ikea = new Silla(4, "blanco", 40, 60);
            Console.WriteLine(ikea.Calculo());
            ikea.MostrarAgradecimiento();
        }
    }
}
```

8. Excepciones

Las excepciones son el mecanismo recomendado en la plataforma .NET para la propagación de errores que se produzcan durante la ejecución de las aplicaciones (divisiones por cero, intentos de lectura de archivos dañados, etc.). Básicamente una excepción es un objeto derivado de **System.Exception** que se genera cuando en tiempo de ejecución se produce algún error y que contiene información sobre el mismo.

En el espacio de nombres System de la BCL hay predefinidas múltiples excepciones derivadas de **System.Exception** que se corresponden con los errores más comunes que pueden surgir durante la ejecución de una aplicación. En la siguiente tabla se recogen algunas:

Tipo de la excepción	**Causa de que se produzca la excepción**
ArgumentException	Pasado argumento no válido (base de excepciones de argumentos)
ArgumentNullException	Pasado argumento nulo
ArgumentOutOfRangeException	
ArrayTypeMistmatchException	Asignación a tabla de elemento que no es de su tipo
COMException	Excepción de objeto COM
DivideByZeroException	División por cero
IndexOutOfRangeException	Índice de acceso a elemento de tabla fuera del rango válido (menor que cero o mayor que el tamaño de la tabla)
InvalidCastException	Conversión explícita entre tipos no válida
InvalidOperationException	Operación inválida en estado actual del objeto
InteropException	Base de excepciones producidas en comunicación con código inseguro
NullReferenceException	Acceso a miembro de objeto que vale null
OverflowException	Desbordamiento dentro de contexto donde se ha de comprobar los desbordamientos (expresión constante, instrucción checked, operanción checked u opción del compilador /checked)
OutOfMemoryException	Falta de memoria para crear un objeto con new
SEHException	Excepción SHE del API Win32
StackOverflowException	Desbordamiento de la pila, generalmente debido a un excesivo número de llamadas recurrentes.
TypeInizializationException	Ha ocurrido alguna excepción al inicializar los campos estáticos o el constructor estático de un tipo. En InnerException se indica cuál es.

8.1 Lanzamiento de excepciones. Instrucción throw

Para informar de un error no basta con crear un objeto del tipo de excepción apropiado, sino también hay que pasárselo al mecanismo de propagación de excepciones del CLR. A esto se le llama lanzar la excepción, y para hacerlo se usa la siguiente instrucción:

```
throw <objetoExcepciónALanzar>;
```

Por ejemplo, para lanzar una excepción de tipo DivideByZeroException se podría hacer:

```
throw new DivideByZeroException();
```

Si el objeto a lanzar vale null, entonces se producirá una NullReferenceException que será lanzada en vez de la excepción indicada en la instrucción throw.

8.2 Captura de excepciones. Instrucción try

Una vez lanzada una excepción es posible escribir código que se encargue de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación aborte, mostrando un mensaje de error en el que se describe la excepción producida y dónde se ha producido.

Así, dado el siguiente código fuente de ejemplo:

```
using System;

class A
{
    public void F()
    {
        G();
    }
    static public void G()
    {
        int c = 0;
        int d = 2 / c;
    }
}
class PruebaExcepciones
{
    static void Main()
    {
        A obj1 = new A();
        obj1.F();
    }
}
```

Al compilarlo no se detectará ningún error ya que al compilador no le merece la pena calcular el valor de c en tanto que es una variable, por lo que no detectará que dividir 2/c no es válido. Sin embargo, al ejecutarlo se intentará dividir por cero en esa instrucción y ello provocará que aborte la aplicación mostrando el siguiente mensaje:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide
by zero. at PruebaExcepciones.Main()
```

Como se ve, en este mensaje se indica que no se ha tratado una excepción de división por cero (tipo DivideByZeroException) dentro del código del método Main() del tipo PruebaExcepciones. Si se desea tratar la excepción hay que encerrar la división dentro de una instrucción try con la siguiente sintaxis:

```
try
<instrucciones>
catch (<excepción1>) <tratamiento1>
catch (<excepción2>) <tratamiento2>
...
finally   <instruccionesFinally>
```

El significado de try es el siguiente: si durante la ejecución de las <instrucciones> se lanza una excepción de tipo <excepción1> (o alguna subclase suya) se ejecutan las instrucciones <tratamiento1>, si fuese de tipo <excepción2> se ejecutaría <tratamiento2>, y así hasta que se encuentre una cláusula catch que pueda tratar la excepción producida. Si no se encontrase ninguna y la instrucción try estuviese anidada dentro de otra, se miraría en los catch de su try padre y se repetiría el proceso. Si en un try no se encuentra un catch compatible, antes de pasar a buscar en su try padre o en su método llamante padre se ejecutarán las <instruccionesFinally>. El bloque finally es opcional, y si se incluye ha de hacerlo tras todas los bloques catch. Las <instruccionesFinally> de este bloque se ejecutarán tanto si se producen excepciones en <instrucciones> como si no.

Así, para tratar la excepción del ejemplo anterior de modo que una división por cero provoque que a la variable d se le asigne el valor 0, se podría reescribir G() de esta otra forma:

```
static public void G()
{
    int c, d;
    try
    {
       c = 0;
       d = 2 / c;
    }
    catch (DivideByZeroException) { d = 0; }
}
```

Para simplificar tanto el compilador como el código generado y favorecer la legibilidad del fuente, en los catchs se busca siempre orden de aparación textual, por lo que para evitar catchs absurdos, no se permite definir catchs que puedan capturar excepciones capturables por catchs posteriores a ellos en su misma instrucción try.
Aunque los bloques catch y finally son opcionales, toda instrucción try ha de incluir al menos un bloque catch o un bloque finally.

El siguiente ejemplo resume cómo funciona la propagación de excepciones:

```
using System;

      class MiException:Exception {}

      class Excepciones
      {
          public static void Metodo()
          {
                  try
                  {
                        Console.WriteLine("En el try de Método()");
                        Método2();
                        Console.WriteLine("Al final del try de
                                           Método()");
                  }
                  catch (OverflowException)
                  {
                        Console.WriteLine("En el catch de Método()");
                  }
                  finally
                  {
                        Console.WriteLine("finally de Método()");
                  }
            }
            public static void Método2()
            {
                  try
                  {
                        Console.WriteLine("En el try de Método2()");
                        throw new MiException();
                        //Console.WriteLine("Al final del try de
                            Método2()");
                  }
                  catch (DivideByZeroException)
                  { Console.WriteLine("En el catch de Método2()"); }
                  finally
                  { Console.WriteLine("finally de Método2()"); }
            }
            public static void Main()
            {
                  try
                  {
                    Console.WriteLine("En el try de Main()");
                    Metodo();
```

```
                    Console.WriteLine("Al final del try de Main()");

                }
                catch (MiException)
                {
                    Console.WriteLine("En el catch de Main()");
                }
                finally
                {
                    Console.WriteLine("finally de Main()");
                }
            }
        }
```

Nótese que en este código, lo único que se hace es definir un tipo nuevo de excepción llamado MiException y llamarse en el **Main()** a un método llamado Método() que llama a otro de nombre Método2() que lanza una excepción de ese tipo. Viendo la salida de este código es fácil ver el recorrido seguido durante la propagación de la excepción:

```
En try de Main()
En try de Método()
En try de Método2()
finally de Método2
finally de Método
En catch de Main()
finally de Main()
```

Como se puede observar, hay muchos **WriteLine()** que nunca se ejecutan ya que en cuanto se lanza una excepción se sigue ejecutando tras la instrucción siguiente al **try** que la trató (aunque ejecutando antes los **finally** pendientes, como se deduce de la salida del ejemplo) De hecho, el compilador se dará cuenta que la instrucción siguiente al **throw** nunca se ejecutará e informará de ello con un mensaje de aviso.

En el catch se puede incluir una variable que nos permitirá acceder a determinados atributos de la excepción, esto se hará de la siguiente manera:

```
catch (MiException e)
{
      Console.WriteLine("En el catch de Main(){0}",e.Message);
}
```

En este caso la salida de este catch nos mostraría el mensaje siguiente:

```
En catch de Main() se produjo una excepción del tipo MiException
```

8.3 Encadenar lanzamientos

Puedo crear un bloque catch para capturar una excepción en un ámbito y añadir un mensaje específico para ese ámbito y posteriormente relanzarla para ser capturada en otro ámbito.

Este proceso se puede repetir de forma sucesiva.

```
try
{
        // ámbito 1
}
catch (ArgumentException e)
{throw new ArgumentException("mensaje específico en ámbito 1", e);}
```

La mayoría de constructores de excepciones de las BCL, admiten una sobre carga con el parámetro Exception innerException (NULL por defecto).
Este me permitirá recorrer todos los objetos excepción en el orden que se han ido relanzando y así acceder a mensajes específicos en cada ámbito.
Ejemplo AVANZADO de anidación de gestión excepciones

```
try {
      // bloque try 1
     try {  // bloque try 2
            …
          try {   // bloque try 3
                …
             }
            catch(E1)
            { throw new E1("Mensaje específico bloque 3", E1);
              // Capturo la excepción y la relanzo para que  sea
             // gestionada del bloque 3 por alguno de los otros
            //  bloques.
            }
       …
    }
    catch(E1) {throw new E1("Mensaje específico bloque 2", E1);}
    catch(E2) { // No puede ser E1 ni una subclase de E1}
    finally { // Se ejecuta siempre aunque relance E1}
 }
 catch(E1) {// Aquí puedo acceder a la última E1 generada y a través
            //   de   InnerException   ir   recorriendo   los   mensajes
específicos
            // añadidos en cada bloque.}
```

8.4 Creando nuestras propias excepciones

Para crear un tipo excepción, deberemos heredar De System.Exception y podemos o no definir nuestros propios constructores e incluir el código necesario.

```
class ExcepcionDepartamento : Exception
{
     public ExcepcionDepartamento(string message) : base(message)
     {

     }
 }
```

En el código de ejemplo hemos creado una excepción ExcepcionDepartamento que usaré para saber cuándo no he controlado algo en los departamentos de mi compañía. Hemos redefinido el constructor al que se le pasa un string, y al llamar al constructor padre este se encargará de asignar esa string a la propiedad Message.

Supongamos el siguiente programa:

```
    enum Departamentos { Contable, Desarrollo, Marketing };
    class Program
    {
       static void ImprimeNominas(Departamentos departamento)
       {
          switch (departamento)
          {
             case Departamentos.Contable:
                Console.WriteLine(" Imprimiendo nóminas
                                                    contabilidad.");
                break;
              case Departamentos.Desarrollo:
                Console.WriteLine("Imprimiendo nóminas
                                                    contabilidad.");
                break;
              default:
                throw new ExcepcionDepartamento(
                     $"No se pueden imprimir nóminas de " +
                     $"este departamento de {departamento}.");
                }
       }
       static void Main()
       {
          try
          {
             ImprimeNominas(Departamentos.Marketing);
          }
          catch (ExcepcionDepartamento e)
          {
             Console.WriteLine(e.Message);
          }
       }
 }
```

Si llamamos al método de imprimir nominas con un departamento erróneo, se lanzará la excepción que hemos creado, que luego capturaremos en la Main y trataremos como corresponda.

5.5. Jerarquía de excepciones en .NET

En el gráfico posterior podemos ver un ejemplo con la jerarquía de las excepciones más usadas. Fíjate que algunas no son recomendables capturarlas, lanzarlas o derivarlas.
Recomendaciones de uso de excepciones estándar:
https://msdn.microsoft.com/es-es/library/ms229007(v=vs.110).aspx

Prácticas de uso recomendadas:
https://msdn.microsoft.com/es-es/library/ms229030(v=vs.110).aspx