C#

Programación orientada a objetos

Índice

Crear un proyecto

Para desarrollar un proyecto debemos:

1. Identificar los diferentes elementos que serán parte de la aplicación (clases).

2. Definir la información que necesitamos almacenar de cada elemento (atributos).

3. Definir las operaciones que cada elemento puede necesitar hacer (métodos).

4. Crear los diferentes objetos de cada clase.

Cuando trabajamos con clases y objetos es normal definir cada clase en su propio archivo fuente.

Elementos

Clases

Las clases son plantillas que categorizan los diferentes elementos de la aplicación.

Atributos

Para cada clase, debemos almacenar cierta información de interés,por ejemplo el nombre o las páginas de un libro.

Objetos

Una vez sabemos los atributos de una clase, podemos crear objetos de la misma

Un objeto es una instancia concreta de una clase,por ejemplo un libro concreto.

Puede haber varios objetos de una clase cuando la aplicación se está ejecutando.

Métodos

Cada objeto de la aplicación puede necesitar hacer ciertas operaciones.

Por ejemplo, los clientes pueden querer buscar libros, o comprarlos.

Estas operaciones se llaman **métodos** de esas clases.

Definición de clases

Usamos la palabra **class** seguida del nombre de la clase (*comenzando en mayúscula normalmente*).

Dentro del código de la clase ubicamos los atributos, también llamados *variables de instancia*.

También podemos definir los métodos como funciones internas de la clase (en este caso sin la palabra **static**, pero con la palabra **public** normalmente).

```
class Libro
{
    string titulo;
    int numPaginas;
    double precio;

    public void MostrarInformacion()
    {
        Console.WriteLine("Información del libro:");
        Console.WriteLine("Título: " + titulo);
        Console.WriteLine("Páginas: " + numPaginas);
        Console.WriteLine("Precio: " + precio);
    }
}
```

Crear objetos

Declaramos una variable del tipo de la clase y empleamos el operador **new** para crear el objeto.

Usamos los métodos de la clase anteponiéndoles el operador **punto (.)**.

```
Libro miLibro = new Libro();
miLibro.MostrarInformacion();
```

Acceder a los atributos

Getters y setters

Métodos especiales que permiten obtener o asignar el valor de cada atributo.

```
class Libro
{
    string titulo;
    int numPaginas;
    double precio;

    public string GetTitulo()
    {
        return titulo;
    }

    public void SetTitulo(string t)
    {
        titulo = t;
    }

    public void MostrarInformacion()
    {
        Console.WriteLine("Información del libro:");
        Console.WriteLine("Título: " + titulo);
        Console.WriteLine("Páginas: " + numPaginas);
        Console.WriteLine("Precio: " + precio);
    }
}

class Principal
{
    static void Main()
    {
        Libro miLibro = new Libro();
        miLibro.SetTitulo("El juego de Ender");
        miLibro.MostrarInformacion();
    }
}
```

Propiedades

Forma alternativa y abreviada de escribir **getters** y **setters** en **C#**.

(No disponible en otros lenguajes).

```
// Alternativa con propiedades
   public string Titulo
   {
      get
      {
         return titulo;
      }

      set
      {
         titulo = value;
      }
   }
   class Principal
{
   static void Main()
   {
      Libro l = new Libro(...);
      l.Titulo = "El juego de Ender";  // set
      Console.WriteLine(l.Titulo);     // get
   }
}
```

Constructores

Son métodos especiales que permiten crear objetos.

Tienen el mismo nombre que la clase a la que pertenecen, pero no tienen tipo de retorno.

En su código normalmente se asignan valores iniciales a los atributos.

Podemos tener tantos constructores como queramos, con distintos números o tipos de parámetros.

Constructor por defecto

No tiene **parámetros**, y se usa para asignar valores por defecto a los atributos.

Se genera automáticamente si la clase no tiene constructor definido.

```
// Constructor por defecto

    public Libro()
    {
        titulo = "";
        numPaginas = 0;
        precio = 0;
    }
```

Constructor parametrizado

Recibe **parámetros** con valores que normalmente se asignan a algún atributo.

```
// Constructor parametrizado

   public Libro(string t, int n, double p)
   {
      titulo = t;
      numPaginas = n;
      precio = p;
   }
```

Destructores

Métodos especiales que se encargan de liberar memoria que hayamos reservado (típicamente para **colecciones dinámicas** o **ficheros**).
No muy habituales en **C#**, más habituales en **C++.**

```
class Libro
{
   ...
   ~Libro()
   {
      // Liberar memoria
      // Cerrar ficheros
   }
}
```

Visibilidad

Establece qué elementos son visibles desde qué partes del código.

Se gestiona a través de los **modificadores de acceso**:

public

elementos accesibles desde cualquier parte del código.

private

elementos accesibles sólo desde dentro de la propia clase.

Los atributos normalmente se etiquetan como privados, y los métodos como públicos.

Si no se indica nada en un elemento, se le asigna visibilidad privada.

```
class Libro
{
    private string titulo;
    private int numPaginas;
    private double precio;
```

Uso de this

Palabra especial para referirnos a cualquier elemento de la clase en la que estamos (**atributo**, **constructor** o **método**).

Útil para diferenciar los elementos de la clase de otros elementos externos que puedan llamarse igual.

Habitual en **parámetros** de **constructores** y **setters**.

```
// Constructor parametrizado

    public Libro(string titulo, int numPaginas, double
precio)
    {
        this.titulo = titulo;
        this.numPaginas = numPaginas;
        this.precio = precio;
    }

// Getters y setters

    public void SetTitulo(string titulo)
    {
        this.titulo = titulo;
    }
```

Arrays de objetos

Definición

Podemos definir un **array** de **objetos** de una **clase** de forma similar a cualquier otro **array** (indicando tipo de dato y nombre de variable).

```
Libro[] libros = new Libro[10];
```

Rellenar el array

Necesitamos instanciar (**new**) cada objeto que será parte del array.

```
libros[0] = new Libro("El juego de Ender", 321, 7.95);
libros[1] = new Libro("La historia interminable", 525, 14.15);
...

// O con un bucle:
for (int i = 0; i < libros.Length; i++)
{
    libros[i] = new Libro(...);
}
```

Relaciones entre clases

Las clases no están sueltas en una aplicación, necesitan **interactuar** entre ellas.

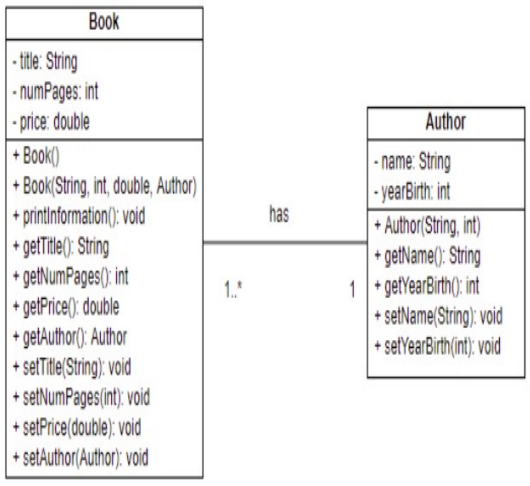
Asociación

Relación entre dos clases donde una es parte de los elementos de la otra.

Un objeto o array de objetos de una clase es un atributo de la otra.

Relación **tiene un**.

Ejemplo: un libro tiene un autor, un autor puede tener muchos libros



```
class Author
{
    private string name;
    private int yearBirth;

    public Author(string name, int yearBirth)
    {
        this.name = name;
        this.yearBirth = yearBirth;
    }

    ... // Getters & setters
}

class Book
{
    private string title;
    private int numPages;
    private double price;
    private Author author;

    public Book(string title, int numPages, double price,
Author author)
    {
        this.title = title;
        this.numPages = numPages;
        this.price = price;
        this.author = author;
    }
```

```
    ... // Getters & setters
}

class BookExample
{
    static void Main()
    {
        Author a = new Author("J.R.R. Tolkien", 1892);

        // The lord of the Rings, 850 pages, 13.50 eur, Tolkien
        Book b = new Book("The lord of the Rings", 850,
13.50, a);

        Console.WriteLine(b.GetTitle());
        Console.WriteLine(b.GetAuthor().GetName());
    }
}
```

Asociaciones y referencias

Si queremos asociar el mismo autor a varios libros, debemos usar el mismo objeto.

```
Author a1 = new Author("J.R.R. Tolkien", 1892);
Author a2 = new Author("J.R.R. Tolkien", 1892);
// a2 no es el mismo que a1, aunque tengan atributos iguales

Book b1 = new Book("The lord of the Rings", 850, 13.50, a1);
Book b2 = new Book("The hobbit", 345, 8.76, a2);
// Autor distinto
Book b3 = new Book("The hobbit", 345, 8.76, a1);
// Mismo autor
```

Navegabilidad

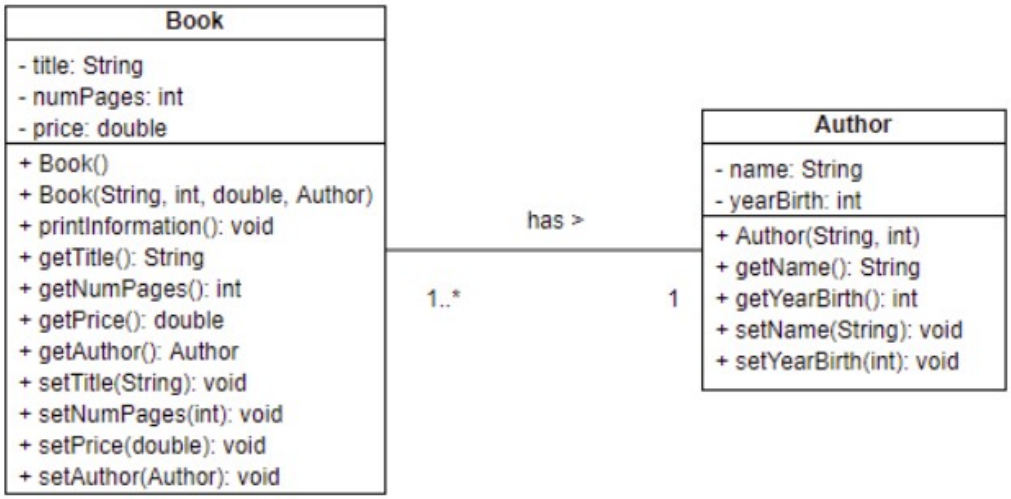
Las asociaciones son (o pueden ser) bidireccionales.

Un libro tiene un autor, y un autor puede tener muchos libros.

Asociación unidireccional

Si queremos que una asociación sea unidireccional, en el diagrama definimos una flecha indicando la dirección de la asociación.

Ejemplo: podemos saber el autor de un libro, pero no los libros de un autor.



Asociación bidireccional

Una casa tiene un salón, y un salón pertenece a una casa

```
class LivingRoom
{
   private int area;
   private House house;

   // Creamos el salón sin asignar la casa
   public LivingRoom(int area)
   {
      this.area = area;
      // La casa no se asigna
   }
}

class House
{
   private string address;
   private int rooms;
   private LivingRoom livingRoom;

   public House(String address, int rooms, LivingRoom livingRoom)
   {
      this.address = address;
      this.rooms = rooms;
      this.livingRoom = livingRoom;
      // Asignamos la casa al salón
      livingRoom.setHouse(this);
   }
}


class Main
{
   static void Main()
   {
      LivingRoom lr = new LivingRoom(40);
      House h = new House("CSharp Street", 3, lr);
      // Aquí la asociación ya es bidireccional
   }
}
```

Composiciones

la parte no puede existir sin el todo.

Implementación

Tenemos que asegurarnos de que la parte sea un elemento **privado del todo**, que no puede ser accedido desde fuera

(De lo contrario podría asignarse a otra cosa).

Además, la parte debe ser **construida dentro del todo**

(Si se pasa construida desde fuera, ese mismo objeto puede reutilizarse).

```
class Coche
{
    private Motor motor;

    public Coche(ParametrosMotor params)
    {
        motor = new Motor(params);
    }
}
```

Agregaciones

la parte puede existir sin el todo.

implementación

Se implementan como asociaciones normales.

Las composiciones normalmente también se implementan así (por lo que no son composiciones en la práctica).

Herencia

Usamos **herencia** cuando queremos crear una clase que tome todos los elementos de otra, incluyendo además sus propios elementos particulares.

Relación de tipo **es un**.

Implementación

Definimos primero la clase principal, también llamada **clase padre**, **superclase** o **clase base**.

Definimos luego la **clase secundaria**, también llamada **clase hija**, **subclase** o **clase derivada**.

Usamos los **dos puntos** (**:)** para indicar de qué clase hereda.

```
class Porton : Puerta
{
    private bool bloqueada;

    public Porton(int ancho, int alto, string color)
    {
        this.ancho = ancho;
        this.alto = alto;
        this.color = color;
    }

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }
}
```

Visibilidad

Una subclase no puede acceder a los elementos privados (**private**) de su clase padre.

Para evitar eso, podemos hacer que esos elementos sean **protected**.

La visibilidad protegida (**protected**) permite a las subclases acceder a esa información.

```
class Puerta
{
    protected int ancho;
    protected int alto;
    protected string color;
    protected bool abierta;

    ... // El resto no cambia
}
```

Modificar métodos de la clase padre

Si un método ya existe en la **clase padre**, podemos definir un nuevo comportamiento en la **clase hija** usando el modificador **new** en el método.

El código del método padre ya no se ejecutará en el hijo, y **se reemplazará por el nuevo**.

```
class Porton : Puerta
{
    private bool bloqueada;

    ...

    public new void MostrarInformacion()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }
}
```

new no sirve cuando definimos variables **polimórficas**, porque **se utiliza el método del padre**.

Es mejor utilizar la palabra virtual en el método de la clase padre y override en el método redefinido en la hija.

Virtual

indica que ese método quizá pueda ser cambiado en las hijas

override

alude a que ese método existe en la clase padre, y está siendo cambiado

```
using System;

class Animal
{
    public virtual void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}

// -----------------

class Perro: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// -----------------

class Gato: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}
```

Constructores y herencia

Cuando creamos constructores en una subclase, lo primero que hacen es llamar a un constructor de la clase base.

Si no indicamos nada, se llama al constructor por defecto.

Polimorfismo

Alude a que un elemento pueda adoptar diferentes formas.

Polimorfismo de sobrecarga

una clase puede tener varios métodos con el mismo nombre y distinto número de parámetros o tipos de retorno.

Polimorfismo puro

un objeto se puede comportar como cualquiera de sus subtipos

```
Animal[] misAnimales = new Animal[3];
misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new GatoSiames();
```

Detectar subtipo en polimorfismo

Por defecto una **variable polimórfica** sólo puede utilizar los métodos de la clase padre.

Podemos detectar de qué **subtipo** concreto es esa variable para **convertirla** (temporalmente) a ese **subtipo** y usar los **métodos de la clase hija** si es necesario.

Usaremos el operador **is**.

```
Animal a = new Perro();
...
if (a is Perro)
{
   ((Perro)a).MetodoDePerro();
}
```

Uso de base

Término para referirnos a **elementos de la clase padre**.

Pueden ser **atributos**, métodos o **constructores**.

Permite, entre otras cosas, redefinir código en las clases hija **apoyándonos** o **reutilizando el código de la clase padre**.

Constructores

```
class Porton : Puerta
{
   private bool bloqueada;

   public Porton(int ancho, int alto, string color)
   : base(ancho, alto, color)
   {
   }

   ... // Resto de métodos
}
```

Métodos

```
class Porton : Puerta
{
    private bool bloqueada;

    ...

    public override void MostrarInformacion()
    {
        base.MostrarInformacion();
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }

    ... // Resto de métodos
}
```

Dependencia

Establece una conexión entre dos clases donde una de ellas utiliza objetos de la otra en algún punto de su código.

```
class Application
{
   ...

   public void AMethod(Window w)
   {
      ...
   }

   public void AnotherMethod()
   {
      Window w = new Window(...);
      ...
   }
}
```

Conceptos avanzados

Static

Delante de una variable, define una variable de clase,Delante de un método, define un método de clase.

Comunes a todos los objetos de una clase, no es necesario crear ningún objeto para usarlos,Para usarlos, se antepone el nombre de la clase al del elemento estático.

```
class Hardware
{
   public static void BorrarPantalla()
   {
      for (byte i = 0; i < 25; i ++)
         Console.WriteLine();
   }

   public static void UnMetodo()
   {
      Console.WriteLine("Pulsa Intro para borrar");
      Console.ReadLine();
      BorrarPantalla();   // Misma clase, no hace falta "Hardware."
      Console.WriteLine("Borrado!");
   }
}

class Principal
{
   static void Main()
   {
      Hardware.BorrarPantalla();
   }
}
```

Restricciones

Desde un método estático no se puede llamar a métodos que no lo sean(Salvo que se cree un objeto de esa clase y se use para llamar a los métodos).

Por tanto, todo lo que utilice un método estático debe ser también estático, o ser algo local al método (parámetros u objetos definidos dentro del método).

Sobrecarga de operadores

C# permite sobrecargar operadores tradicionales, como los aritméticos, para extender su significado a otros ámbitos

(Por ejemplo, sumar objetos complejos que hayamos creado nosotros).

Definimos un método estático llamado **operator**, seguido del operador a redefinir, y que devolverá un objeto del mismo tipo con el que estamos trabajando.

```
class Vector2D
{
   private double x;
   private double y;

   public Vector2D(double x, double y)
   {
      this.x = x;
      this.y = y;
   }


   ...
   public static Vector2D operator + (Vector2D v1,
Vector2D v2)
   {
      Vector2D v3 = new Vector2D(v1.x + v2.x, v1.y +
v2.y);
      return v3;
   }
}

class Principal
{
   static void Main()
   {
      Vector2D v1 = new Vector2D(0, 6);
      Vector2D v2 = new Vector2D(-1, 4.5);
      Vector2D v3 = v1 + v2;
      Console.WriteLine(v3);
   }
}
```

Clases abstractas

Clases que no pueden ser instanciadas directamente, por considerarse incompletas.

Pueden tener parte de su código sin implementar (**métodos abstractos**).

Utilidad: definir un código base que luego amplíen o completen las clases hija.

```
abstract class Animal
{
   protected string nombre;

   public Animal(string nombre)
   {
      this.nombre = nombre;
   }

   public abstract void Hablar();
```

Herencia de clases abstractas

```
class Perro : Animal
{
    public Perro(string nombre) : base(nombre)
    {
    }

    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}
```

Clases abstractas a partir de otras

```
abstract class Ave : Animal
{
    public Ave(string nombre) : base (nombre)
    {
    }

    public abstract void Volar();
}

class Pato : Ave
{
    public Pato(string nombre) : base (nombre)
    {
    }

    public override void Hablar()
    {
        Console.WriteLine("Cuac Cuac!");
    }

    public override void Volar()
    {
        Console.WriteLine("Estoy volando");
    }
}
```

Interfaces

Permite definir un conjunto de métodos por implementar,útil para obligar a ciertas clases a que tengan disponibles esos métodos.

Hace que las clases se comporten como un elemento del tipo de la **interfaz**.

Se definen con la palabra **interface** en lugar de **class**.

En C# sólo podemos heredar de una clase, pero implementar muchas interfaces.

```
interface IFiguraGeometrica
{
    double CalcularArea();
    double CalcularPerimetro();
}
```

```
class Cuadrado : IFiguraGeometrica
{
    private double lado;

    public Cuadrado(double lado)
    {
        this.lado = lado;
    }

    public double CalcularArea()
    {
        return lado * lado;
    }

    public double CalcularPerimetro()
    {
        return 4 * lado;
    }
}
```

Implementar más de una interfaz

```
interface IDibujable
{
    void Dibujar();
}

class Cuadrado : IFiguraGeometrica, IDibujable
{
    private double lado;

    public Cuadrado(double lado)
    {
        this.lado = lado;
    }

    public double CalcularArea()
    {
        return lado * lado;
    }

    public double CalcularPerimetro()
    {
        return 4 * lado;
    }

    public void Dibujar()
    {
        // Dibujo de un cuadrado
    }
}
```

Heredar e implementar interfaces

Primero se indica la clase de la que se hereda, y luego las interfaces que se implementan, todo separado por comas.

```
class Cuadrado : ClaseAHeredar, IFiguraGeometrica, IDibujable
{
    ...
}
```

Ordenación

Icomparable

C# tiene una interfaz llamada **IComparable** que permite establecer **criterios de comparación entre objetos de una clase**.

Permite utilizar **Array.Sort** con un criterio que ordene automáticamente un conjunto de objetos.

Basta con hacer que la clase a ordenar implemente esa interfaz y redefina su método **CompareTo** definiendo el criterio de comparación.

```
class Persona : IComparable<Persona>
{
   private string nombre;
   private int edad;

   public string Nombre
   {
      get { return nombre; }
      set { nombre = value; }
   }
   public int Edad
   {
      get { return edad; }
      set { edad = value; }
   }
   public Persona(string nombre, int edad)
   {
      this.nombre = nombre;
      this.edad = edad;
   }

   public override string ToString()
   {
      return nombre + " (" + edad + " años)";
   }
   public int CompareTo(Persona otra)
   {
      return this.nombre.CompareTo(otra.nombre);
   }
}
```

```
class Principal
{
   static void Main()
   {
      Persona[] personas = new Personas[4];
      personas[0] = new Persona("Nacho", 43);
      personas[1] = new Persona("Ana", 39);
      personas[2] = new Persona("Juan", 70);
      personas[3] = new Persona("Mario", 8);

      Array.Sort(personas);

      foreach(Persona p in personas)
         Console.WriteLine(p);
   }
}
```

Ordenamos personas por su nombre. El método CompareTo se puede usar para comparar datos simples (enteros, reales), igual que cadenas.

Icomparer

Otra interfaz que permite establecer el criterio de ordenación entre los objetos de una clase.

No se implementa en la clase a ordenar, sino en otra diferente.

Método **Compare**, que recibe dos objetos del tipo a comparar,necesitamos incluir **System.Collections.Generic**.

Ventajas

No hace falta tocar el código de la clase afectada.

Podemos definir múltiples criterios de comparación.

```
// Nueva clase que implementa la interfaz
class ComparadorPersonas: IComparer<Persona>
{
    public int Compare(Persona p1, Persona p2)
    {
        return p1.Nombre.CompareTo(p2.Nombre);
    }
}

// Clase Persona (sin interfaz)
class Persona
{
    private string nombre;
    private int edad;

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }

    public int Edad
    {
        get { return edad; }
        set { edad = value; }
    }
```

```
    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public override string ToString()
    {
        return nombre + " (" + edad + " años)";
    }
}

class Principal
{
    static void Main()
    {
        Persona[] personas = new Persona[4];
        personas[0] = new Persona("Nacho", 43);
        personas[1] = new Persona("Ana", 39);
        personas[2] = new Persona("Juan", 70);
        personas[3] = new Persona("Mario", 8);

        Array.Sort(personas, new ComparadorPersonas());

        foreach(Persona p in personas)
            Console.WriteLine(p);
    }
}
```

Clases selladas

Clase de la que no se puede heredar,se definen con la palabra **sealed**.

String es una clase sellada.

```
sealed class MiClase { ... }
```

Ventajas

mejoran la velocidad de ejecución, al no tener que comprobar opciones **polimórficas** ni posibles conversiones a **subtipos**.

ToString

El método **ToString** muestra los campos que denominemos de un objeto de la clase en el que se encuentra el método

```
public override string ToString()
    {
        return nombre + " (" + edad + " años)";
    }
```