OOP - C#

Jorge Mario Londoño Peláez & Varias AI

May 7, 2025

1 Repaso Conceptos OOP - Version C#

1.1 Objetivos de Aprendizaje

Al finalizar este módulo, serás capaz de:

- Comprender los fundamentos de la Programación Orientada a Objetos
- Implementar clases y objetos en C#
- Aplicar los principios de encapsulamiento, herencia y polimorfismo
- Utilizar propiedades, eventos y delegados
- Manejar excepciones en programas orientados a objetos
- Implementar patrones de diseño básicos

1.2 Qué es la programación orientada a objetos

La Programación Orientada a Objetos (OOP) es un paradigma de programación que se basa en el concepto de "objetos", los cuales son entidades que combinan datos (atributos) y comportamiento (métodos). A diferencia de la programación procedimental, que se centra en funciones o procedimientos, la OOP organiza el código en torno a objetos que interactúan entre sí. En comparación con la programación funcional, que se enfoca en la evaluación de funciones y la inmutabilidad, la OOP permite modelar el mundo real de manera más intuitiva mediante la creación de objetos que representan entidades del problema.

1.3 Clases y Objetos

En C#, una **clase** es una plantilla o un plano para crear objetos. Define las características (atributos) y los comportamientos (métodos) que tendrán los objetos de esa clase. Un **objeto** es una instancia específica de una clase.

La palabra clave `this` se utiliza dentro de un método de instancia para referirse al objeto actual. Permite acceder a los miembros de la clase desde dentro de la misma.

1.3.1 Ejemplo: Definición e instancias de una Clase

```
    public class Coche {
        private string marca;
        private string modelo;

        public Coche(string marca, string modelo) {
            this.marca = marca;
            this.modelo = modelo;
        }

        public string Marca {
            get { return marca; }
            set { marca = value; }
        }

        public string Modelo {
            get { return modelo; }
            set { modelo = value; }
        }

        public void Acelerar() {
            Console.WriteLine("El coche " + marca + " " + modelo + " esta
    acelerando.");
        }
    }

public class Ejemplo {
    public static void Main(string[] args) {
        Coche coche1 = new Coche("Toyota", "Corolla");
        Coche coche2 = new Coche("Honda", "Civic");

        coche1.Acelerar(); // Output: El coche Toyota Corolla esta acelerando.
        coche2.Acelerar(); // Output: El coche Honda Civic esta acelerando.
    }
}
```

Listing 1: Ejemplo de clase y objetos en C#

1.3.2 Variables y métodos estáticos

Las variables y métodos estáticos pertenecen a la clase en sí, en lugar de a una instancia específica de la clase. Se acceden utilizando el nombre de la clase en lugar del nombre de un objeto.

```
public class Contador {
    public static int conteo = 0;

    public static void Incrementar() {
        conteo++;
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
        Contador.Incrementar();
        Console.WriteLine(Contador.conteo); // Output: 1
        Contador.Incrementar();
        Console.WriteLine(Contador.conteo); // Output: 2
```

```
    }
}
```

Listing 2: Ejemplo de variables y métodos estáticos en C#

1.4 Variables y métodos estáticos

1.5 Encapsulamiento

El encapsulamiento es el principio de ocultar los detalles internos de un objeto y exponer solo la información necesaria. En C#, se logra mediante el uso de modificadores de acceso como `public`, `private`, `protected` e `internal`. Esto permite controlar cómo se accede y se modifica el estado de un objeto, protegiéndolo de manipulaciones no deseadas.

1.6 Herencia

La herencia es un mecanismo que permite crear nuevas clases (subclases o clases derivadas) basadas en clases existentes (superclases o clases base). Las subclases heredan los atributos y métodos de sus superclases, lo que fomenta la reutilización de código y la creación de jerarquías de clases.

1.6.1 Ejemplo de Herencia en C#

```
public class Animal {
    protected string nombre;

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

    public virtual void HacerSonido() {
        Console.WriteLine("Sonido generico de animal");
    }
}

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

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

public class Gato : Animal {
    public Gato(string nombre) : base(nombre) { }

    public override void HacerSonido() {
        Console.WriteLine("Miau!");
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
        Animal animal = new Animal("Animal");
        Perro perro = new Perro("Firulais");
        Gato gato = new Gato("Michi");

```

```
        animal.HacerSonido(); // Output: Sonido generico de animal
        perro.HacerSonido();  // Output: Guau!
        gato.HacerSonido();   // Output: Miau!
    }
}
```

Listing 3: Ejemplo de herencia en C#

1.6.2 La palabra clave base

La palabra clave `base` se utiliza en una clase derivada para acceder a los miembros de la clase base (superclase). Se usa principalmente para:

- Llamar al constructor de la clase base desde el constructor de la clase derivada. Esto es útil para inicializar los miembros de la clase base.
- Acceder a miembros (métodos, propiedades, campos) de la clase base que han sido ocultados en la clase derivada mediante la declaración de miembros con el mismo nombre.
- Pasar argumentos al constructor de la clase base.

Ejemplo:

```
public class Animal {
    public string nombre;

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

    public virtual void HacerSonido() {
        Console.WriteLine("Sonido generico de animal");
    }
}

public class Perro : Animal {
    public string raza;

    public Perro(string nombre, string raza) : base(nombre) {
        this.raza = raza;
    }

    public override void HacerSonido() {
        Console.WriteLine("Guau! Mi nombre es " + nombre + " y soy un " + raza);
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
        Perro perro = new Perro("Firulais", "Labrador");
        perro.HacerSonido();  // Output: Guau! Mi nombre es Firulais y soy un
    Labrador
    }
}
```

Listing 4: Ejemplo del uso de la palabra clave `base` en C#

1.7 Clases Abstractas

Una clase abstracta es una clase que no se puede instanciar directamente. Se utiliza como una plantilla para otras clases, y puede contener métodos abstractos (métodos sin implementación). Las clases que heredan de una clase abstracta deben implementar todos sus métodos abstractos.

1.7.1 Ejemplo: Definición de superclase abstracta y subclase concreta

```
public abstract class Figura {
    public abstract double CalcularArea();
}

public class Circulo : Figura {
    public double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    public override double CalcularArea() {
        return Math.PI * radio * radio;
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
        Circulo circulo = new Circulo(5);
        Console.WriteLine("Area del circulo: " + circulo.CalcularArea()); //
    Output: Area del circulo: 78.53981633974483
    }
}
```

Listing 5: Ejemplo de clase abstracta en C#

1.8 Polimorfismo

El polimorfismo es la capacidad de un objeto de tomar muchas formas. En C#, se logra mediante la herencia y la implementación de interfaces. Permite tratar objetos de diferentes clases de manera uniforme, siempre y cuando compartan una interfaz común.

1.8.1 Ejemplo de Polimorfismo en C#

```
public interface IFigura {
    double CalcularArea();
}

public class Rectangulo : IFigura {
    public double ancho;
    public double alto;

    public Rectangulo(double ancho, double alto) {
        this.ancho = ancho;
        this.alto = alto;
    }

    public double CalcularArea() {
```

```
        return ancho * alto;
    }
}

public class Triangulo : IFigura {
    public double baseTriangulo;
    public double altura;

    public Triangulo(double baseTriangulo, double altura) {
        this.baseTriangulo = baseTriangulo;
        this.altura = altura;
    }

    public double CalcularArea() {
        return 0.5 * baseTriangulo * altura;
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
        IFigura[] figuras = new IFigura[2];
        figuras[0] = new Rectangulo(5, 10);
        figuras[1] = new Triangulo(4, 6);

        foreach (IFigura figura in figuras) {
            Console.WriteLine("Area: " + figura.CalcularArea());
        }
        // Output: Area: 50
        // Output: Area: 12
    }
}
```

Listing 6: Ejemplo de polimorfismo en C#

1.9 Interfaces

Una interfaz es un contrato que define un conjunto de métodos que una clase debe implementar. A diferencia de las clases abstractas, las interfaces no pueden contener ninguna implementación de métodos (antes de la versión 8.0). A partir de la versión 8.0, la interfaz puede proporcionar una implementación predeterminada del método.

Una clase puede implementar múltiples interfaces.

1.9.1 Clases Abstractas vs. Interfaces

Las clases abstractas y las interfaces son mecanismos para lograr la abstracción en C#, pero presentan diferencias clave:

- **Implementación:** Una clase abstracta puede proporcionar una implementación parcial (métodos concretos) además de métodos abstractos, mientras que una interfaz solo define métodos (a partir de C# 8.0, las interfaces pueden tener métodos con una implementación predeterminada, pero su propósito principal sigue siendo la definición de un contrato).
- **Herencia Múltiple:** Una clase solo puede heredar de una única clase abstracta, pero puede implementar múltiples interfaces.

- **Miembros:** Una interfaz solo puede contener declaraciones de métodos, propiedades, eventos e indexadores. Una clase abstracta puede contener campos, constructores, destructores y otros miembros.
- **Modificadores de Acceso:** Los miembros de una interfaz son implícitamente públicos y no pueden tener modificadores de acceso. Los miembros de una clase abstracta pueden tener cualquier modificador de acceso.

Cuándo usar una clase abstracta:

- Cuando existe una relación "es-un" fuerte entre la clase base y las clases derivadas.
- Cuando se desea proporcionar una implementación predeterminada para algunos métodos que las clases derivadas pueden heredar o anular.
- Cuando se necesita utilizar campos o constructores.

Cuándo usar una interfaz:

- Cuando se desea definir un contrato que múltiples clases no relacionadas pueden implementar.
- Cuando se necesita herencia múltiple.
- Cuando se desea lograr un acoplamiento flexible entre clases.

1.9.2 Ejemplo: Definición e implementación de una interfaz en C#

```
1  public interface ITransporte {
2      void Arrancar();
3      void Detener();
4  }
5
6  public class Coche : ITransporte {
7      public void Arrancar() {
8          Console.WriteLine("El coche ha arrancado.");
9      }
10
11     public void Detener() {
12         Console.WriteLine("El coche se ha detenido.");
13     }
14 }
15
16 public class Bicicleta : ITransporte {
17     public void Arrancar() {
18         Console.WriteLine("La bicicleta ha comenzado a rodar.");
19     }
20
21     public void Detener() {
22         Console.WriteLine("La bicicleta se ha detenido.");
23     }
24 }
25
26 public class Ejemplo {
27     public static void Main(string[] args) {
28         ITransporte coche = new Coche();
29         ITransporte bicicleta = new Bicicleta();
```

```
30
31          coche.Arrancar();     // Output: El coche ha arrancado.
32          bicicleta.Arrancar(); // Output: La bicicleta ha comenzado a rodar.
33      }
34  }
```

Listing 7: Ejemplo de interfaz en C#

1.10 Eventos y delegados

1.11 Excepciones

Las excepciones son errores que ocurren durante la ejecución de un programa. C# proporciona mecanismos para manejar excepciones utilizando bloques `try-catch-finally`.

1.11.1 Atrapar excepciones

El bloque `try` contiene el código que puede generar una excepción. El bloque `catch` se utiliza para atrapar y manejar la excepción. El bloque `finally` se ejecuta siempre, independientemente de si se produjo una excepción o no.

```
1  public class Ejemplo {
2      public static void Main(string[] args) {
3          try {
4              int resultado = 10 / 0; // Esto generara una excepcion
       DivideByZeroException
5          } catch (DivideByZeroException ex) {
6              Console.WriteLine("Error: Division por cero. " + ex.Message);
7          } finally {
8              Console.WriteLine("Bloque finally ejecutado.");
9          }
10     }
11 }
```

Listing 8: Ejemplo de manejo de excepciones en C#

1.11.2 Crear excepciones

Se pueden crear excepciones personalizadas heredando de la clase `Exception` o de alguna de sus subclases.

```
1  public class MiExcepcion : Exception {
2      public MiExcepcion(string mensaje) : base(mensaje) { }
3  }
```

Listing 9: Ejemplo de creación de excepción personalizada en C#

1.11.3 Lanzar excepciones

Se utiliza la palabra clave `throw` para lanzar una excepción.

```
1  public class Ejemplo {
2      public static void Main(string[] args) {
3          try {
4              throw new MiExcepcion("Esta es mi excepcion personalizada.");
5          } catch (MiExcepcion ex) {
```

```
            Console.WriteLine("Excepcion atrapada: " + ex.Message);
        }
    }
}
```

Listing 10: Ejemplo de lanzamiento de excepción en C#

1.12 Clases internas o anidadas

Una clase interna (o clase anidada) es una clase que se define dentro de otra clase. A diferencia de Java, en C# las clases internas no tienen acceso directo a las variables de instancia de la clase contenedora. Para acceder a los miembros de la clase contenedora, la clase interna necesita una referencia explícita a una instancia de la clase externa.

Los casos de uso típicos para clases internas en C# incluyen:

- Implementación de patrones de diseño como Builder o Factory Method
- Agrupación lógica de clases relacionadas
- Encapsulamiento de implementaciones específicas que solo son relevantes para la clase contenedora
- Creación de tipos auxiliares que solo tienen sentido en el contexto de la clase principal

1.12.1 Ejemplo de una clase interna

```
public class ClaseExterna {
    private int datoExterno = 10;

    public class ClaseInterna {
        private ClaseExterna claseExterna;

        public ClaseInterna(ClaseExterna claseExterna) {
            this.claseExterna = claseExterna;
        }

        public void MostrarDato() {
            Console.WriteLine("Dato externo: " + claseExterna.datoExterno);
        }
    }

    public static void Main(string[] args) {
        ClaseExterna externa = new ClaseExterna();
        ClaseInterna interna = new ClaseInterna(externa);
        interna.MostrarDato(); // Output: Dato externo: 10
    }
}
```

Listing 11: Ejemplo de clase interna en C#

1.12.2 Ejemplo de uso en un patrón Builder

```
public class Pizza {
    private string masa;
    private string salsa;
    private List<string> ingredientes;

    private Pizza() {
        ingredientes = new List<string>();
    }

    public class Builder {
        private Pizza pizza;

        public Builder() {
            pizza = new Pizza();
        }

        public Builder ConMasa(string masa) {
            pizza.masa = masa;
            return this;
        }

        public Builder ConSalsa(string salsa) {
            pizza.salsa = salsa;
            return this;
        }

        public Builder AgregarIngrediente(string ingrediente) {
            pizza.ingredientes.Add(ingrediente);
            return this;
        }

        public Pizza Construir() {
            return pizza;
        }
    }
}
```

Listing 12: Ejemplo de clase interna en un patrón Builder

2 Diagramas y Visualizaciones

2.1 Diagramas de Clases

Los diagramas de clases son una herramienta fundamental para visualizar la estructura de un sistema orientado a objetos. A continuación se muestran algunos ejemplos:

```
@startuml
class Animal {
    -nombre: string
    +HacerSonido(): void
}

class Perro {
    -raza: string
    +HacerSonido(): void
}

class Gato {
```

```
    -color: string
    +HacerSonido(): void
}

Animal <|-- Perro
Animal <|-- Gato
@enduml
```

Listing 13: Ejemplo de diagrama de clases en formato PlantUML

2.2 Relaciones entre Clases

En OOP, las clases pueden tener diferentes tipos de relaciones:

- **Herencia (is-a)**: Una clase hereda de otra (ej: Perro es un Animal)
- **Composición (has-a)**: Una clase contiene instancias de otra (ej: Car tiene un Engine)
- **Asociación (uses-a)**: Una clase utiliza otra (ej: Student usa Book)
- **Agregación (part-of)**: Una clase es parte de otra (ej: Wheel es parte de Car)

3 Ejercicios de Repaso

3.1 Ejercicios Básicos

1. **Libro y Biblioteca**
   - Crea una clase `Libro` con propiedades para `Titulo`, `Autor`, `ISBN` y `Precio`.
   - Implementa validación en las propiedades usando auto-implemented properties.
   - Crea una clase `Biblioteca` que pueda almacenar múltiples libros y realizar búsquedas.
   - Implementa `ToString()` y `Equals()` para las clases.
2. **Sistema de Empleados**
   - Implementa una jerarquía de clases para diferentes tipos de empleados:
     - `Empleado` (clase base con nombre, salario base)
     - `Desarrollador` (con especialidad y nivel)
     - `Gerente` (con departamento y bonificación)
   - Implementa el cálculo de salario usando propiedades.
   - Utiliza métodos virtuales y override para comportamientos específicos.
   - Implementa `IComparable` para ordenar empleados.
3. **Sistema de Vehículos**
   - Crea una interfaz `IVehiculo` con métodos como `Arrancar()`, `Detener()` y `ObtenerVelocidad()`.
   - Implementa esta interfaz en clases como `Coche`, `Bicicleta` y `Motocicleta`.
   - Utiliza eventos para notificar cambios de estado.
   - Implementa `IDisposable` para limpieza de recursos.

3.2 Ejercicios Intermedios

1. **Sistema de Reservas**

   - Crea un sistema de reservas para un hotel con:
     - `Habitacion` (número, tipo, precio)
     - `Cliente` (datos personales, historial de reservas)
     - `Reserva` (fechas, habitación, cliente)
   - Implementa manejo de excepciones personalizadas.
   - Utiliza interfaces para definir contratos de servicio.
   - Implementa validación de datos usando Data Annotations.

2. **Red Social Simple**

   - Implementa un sistema básico de red social con:
     - `Usuario` (perfil, amigos, publicaciones)
     - `Publicacion` (contenido, fecha, likes)
     - `Comentario` (texto, autor, fecha)
   - Utiliza colecciones genéricas para manejar las relaciones.
   - Implementa eventos para notificar nuevas publicaciones.
   - Utiliza LINQ para consultas complejas.

3.3 Ejercicios Avanzados

1. **Sistema de Gestión de Proyectos**

   - Crea un sistema para gestionar proyectos con:
     - `Proyecto` (nombre, fecha inicio/fin, presupuesto)
     - `Tarea` (descripción, estado, asignado)
     - `Equipo` (miembros, roles)
   - Implementa el patrón Observer usando eventos.
   - Utiliza el patrón Factory para crear diferentes tipos de tareas.
   - Implementa logging usando NLog o Serilog.

2. **Simulador de Banco**

   - Implementa un sistema bancario con:
     - `Cuenta` (abstracta) con subclases `CuentaCorriente` y `CuentaAhorro`
     - `Cliente` (con múltiples cuentas)
     - `Transaccion` (depósitos, retiros, transferencias)
   - Implementa manejo de excepciones personalizadas.
   - Utiliza el patrón Singleton para el registro de transacciones.
   - Implementa validación de datos usando Data Annotations.

3.4 Consejos para los Ejercicios

- Utiliza las características modernas de C# como propiedades auto-implementadas.
- Implementa pruebas unitarias usando NUnit o xUnit.
- Documenta tus clases y métodos usando comentarios XML.
- Utiliza expresiones lambda y LINQ cuando sea apropiado.
- Aprovecha las características específicas de C# como eventos y delegados.

4 Buenas prácticas

4.1 Principios SOLID en C#

Los principios SOLID son fundamentales en el desarrollo de software:

- **S**ingle Responsibility Principle (SRP): Una clase debe tener una única razón para cambiar.
- **O**pen/Closed Principle (OCP): Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación.
- **L**iskov Substitution Principle (LSP): Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases.
- **I**nterface Segregation Principle (ISP): Los clientes no deben depender de interfaces que no utilizan.
- **D**ependency Inversion Principle (DIP): Los módulos de alto nivel no deben depender de módulos de bajo nivel.

4.2 Patrones de Diseño en C#

- **Singleton**: Implementado usando propiedades estáticas y constructores privados.
- **Factory**: Utilizando interfaces y clases abstractas.
- **Observer**: Implementado con eventos y delegados.
- **Strategy**: Utilizando interfaces y delegados.

4.3 Depuración de Código OOP en C#

- Utiliza el depurador de Visual Studio.
- Implementa logging usando NLog o Serilog.
- Escribe pruebas unitarias para verificar el comportamiento.
- Utiliza herramientas de análisis de código como SonarQube.

5 Referencias adicionales

Microsoft Learn: Programación orientada a objetos (C#)
Dev.co: Explorando los Fundamentos de la Programación Orientada a Objetos en C#