# Escuelita Crombie 3er Edición

# Interfaces

Una interfaz es un tipo de entidad que define un conjunto de métodos y propiedades que una clase concreta debe implementar.

En otras palabras, una interfaz especifica un contrato que las clases que la implementan deben cumplir.

Las interfaces en C# permiten lograr la abstracción y la implementación de múltiples herencias, ya que una clase puede implementar varias interfaces.

**OBJETIVO**:

El principal objetivo de utilizar interfaces en nuestro código es desacoplar las diferentes partes del sistema, lo que significa evitar que las clases dependan directamente unas de otras. Esto facilita el control del flujo del código y promueve la reutilización y la escalabilidad.

# Inyección de dependencias

La inyección de dependencias es una técnica de programación en C# que consiste en pasar las dependencias de un objeto a otro, en lugar de que la clase las configure directamente. Esta técnica ayuda a mejorar la modularidad, reutilización y flexibilidad del código, y a hacer que sea más limpio y de calidad.

### 1. Facilita el mantenimiento y la escalabilidad

- **Reducción del acoplamiento:** Las clases no dependen directamente de implementaciones concretas, sino de abstracciones (interfaces), lo que facilita realizar cambios en las implementaciones sin afectar el código que las usa.

- **Actualizaciones más sencillas:** Puedes reemplazar fácilmente una dependencia por una versión actualizada o diferente sin necesidad de modificar la clase que la consume.

### 2. Promueve la reutilización de código

- Las dependencias se pueden compartir entre múltiples clases, lo que permite evitar la duplicación de código y fomenta un diseño más modular.

### 3. Fomenta el diseño basado en interfaces

- Al depender de abstracciones (interfaces) en lugar de implementaciones concretas, el código se vuelve más flexible y adaptable a cambios futuros.

### 4. Mejoras en las pruebas unitarias

- **Facilidad para realizar pruebas:** Permite inyectar dependencias simuladas o "mock" durante las pruebas, lo que elimina la necesidad de utilizar dependencias reales (como bases de datos o servicios externos) que pueden ser lentas, inestables o difíciles de configurar.
- **Aislamiento:** Ayuda a probar unidades individuales sin preocuparte por el comportamiento de sus dependencias.

### 5. Organización del código más limpia

- Elimina la necesidad de inicializar dependencias dentro de las clases, reduciendo el desorden en el código y mejorando su legibilidad.
- Los constructores de las clases reciben explícitamente todas las dependencias necesarias, haciendo que el contrato de uso sea más claro.

### 6. Facilita la gestión de ciclos de vida de objetos

- Con el uso de contenedores de inyección de dependencia (como Microsoft.Extensions.DependencyInjection, Autofac, o Unity), es fácil

gestionar el ciclo de vida de los objetos (transitorio, singleton, scoped), optimizando el uso de recursos.

### 7. Reduce la dependencia de servicios estáticos

- Minimiza el uso de dependencias estáticas o globales que dificultan el mantenimiento y las pruebas.

### 8. Simplifica la configuración y configuración centralizada

- Un contenedor de DI centraliza la configuración de las dependencias de la aplicación, haciendo que los puntos de inicialización y configuración sean fáciles de ubicar y modificar.

## Inyección de dependencias

Para configurar la inyección de dependencias en C#, se puede seguir el siguiente procedimiento:

- Definir las interfaces para las dependencias
- Proporcionar implementaciones concretas
- Utilizar la inyección para pasar las dependencias a las clases
- Utilizar un contenedor de inyección de dependencias

IProductService.cs

```
4 referencias
public interface IProductService
{
    2 referencias
    void CreateProduct(ProductEntity product);
    1 referencia
    void DeleteProduct(ProductEntity product, int idProduct);
    2 referencias
    ProductEntity GetProduct(int idProduct);
    2 referencias
    public List<ProductEntity> GetProducts();
    2 referencias
    void UpdateProduct(ProductEntity product, int idProduct);
}
```

ProductService.cs

```
2 referencias
public class ProductService : IProductService
{
    private readonly StoreDbContext _context;

    0 referencias
    public ProductService(StoreDbContext context)...

    2 referencias
    public void CreateProduct(ProductEntity product)...

    2 referencias
    public void UpdateProduct(ProductEntity product, int idProduct)...

    1 referencia
    public void DeleteProduct(ProductEntity product, int idProduct)...

    2 referencias
    public ProductEntity GetProduct(int idProduct)...

    2 referencias
    public List<ProductEntity> GetProducts()...
}
```

ProductController

```
public class ProductController : ControllerBase
{
    private readonly IProductService _productService;

    0 referencias
    public ProductController(IProductService productService)
    {
        this._productService = productService;
    }
```

Program.cs

```
builder.Services.AddScoped<IProductService, ProductService>();
```

# Principios S.O.L.I.D.

## 1. S - Single Responsibility Principle (Principio de Responsabilidad Única)

**Cada clase debe tener una sola razón para cambiar.**

- Esto significa que una clase debe encargarse de una sola cosa o responsabilidad.
- Si una clase hace demasiadas cosas, es difícil de mantener y probar.

**Ejemplo:** En lugar de una clase que maneje tanto la lógica de facturación como la impresión de facturas, divide las responsabilidades:

## 2. O - Open/Closed Principle (Principio Abierto/Cerrado)

**El código debe estar abierto para extensión, pero cerrado para modificación.**

- Puedes agregar funcionalidades nuevas sin modificar el código existente, evitando errores en código que ya funciona.

**Ejemplo:** Usa interfaces o clases base para permitir extensiones:

## 3. L - Liskov Substitution Principle (Principio de Sustitución de Liskov)

**Una clase derivada debe ser sustituible por su clase base.**

- Los objetos de una clase hija deben poder usarse en lugar de la clase padre sin alterar el comportamiento del programa.

## 4. I - Interface Segregation Principle (Principio de Segregación de Interfaces)

**Una clase no debe verse obligada a implementar interfaces que no utiliza.**

- Divide las interfaces grandes en varias interfaces pequeñas y específicas.

**Ejemplo:** En lugar de una interfaz gigante, divide la interfaz en responsabilidades específicas

## 5. D - Dependency Inversion Principle (Principio de Inversión de Dependencias)

**Las clases deben depender de abstracciones, no de implementaciones concretas.**

- Esto se logra utilizando interfaces o clases abstractas para desacoplar dependencias.

**Ejemplo:** En lugar de depender de una implementación concreta, usa una abstracción/interfaz