**Programación Orientada a Objetos (POO)**

La **Programación Orientada a Objetos (POO)** es una forma de organizar y estructurar el código en donde los datos y las funciones que operan sobre esos datos se agrupan en **objetos**. La idea central es que se puede representar cualquier cosa (una persona, un coche, una cuenta bancaria) como un objeto en el programa. Esto hace que el código sea más fácil de mantener y reutilizar.

**Ejemplo:**

Imagina un programa que simula una tienda. En lugar de tener variables sueltas para cada artículo, cliente o pedido, puedes crear **objetos** que representan estos elementos. Por ejemplo, puedes tener un objeto Articulo con atributos como nombre y precio, y métodos como calcularDescuento().

**Clase**

Una **clase** es como un plano o molde para crear objetos. Define las características (atributos) y comportamientos (métodos) que los objetos creados a partir de ella tendrán. No es el objeto en sí, sino la definición de cómo se verá ese objeto.

**Ejemplo:**

Piensa en la clase Coche. Esta clase puede tener atributos como marca, modelo y color, y métodos como arrancar() y frenar(). A partir de esa clase, puedes crear muchos coches (objetos) con diferentes marcas, modelos y colores.

A computer screen with text on it

Description automatically generated

**Objeto**

Un **objeto** es una instancia de una clase. Si la clase es el plano, el objeto es el producto final. Cada objeto tiene sus propios valores para los atributos definidos en la clase.

**Ejemplo:**

A partir de la clase Coche, puedes crear objetos como coche1 y coche2, cada uno con sus propios valores para marca, modelo y color.

A screen shot of a computer code

Description automatically generated

**Herencia**

La **herencia** es una característica que permite crear una nueva clase a partir de una clase existente, heredando sus atributos y métodos. Esto es útil para evitar repetir código.

**Ejemplo:**

Si tienes una clase Vehiculo, puedes crear una clase Coche que herede de Vehiculo para reutilizar características comunes (como numeroDeRuedas), y luego agregar características específicas de un coche (como marca y modelo).

A screen shot of a computer code

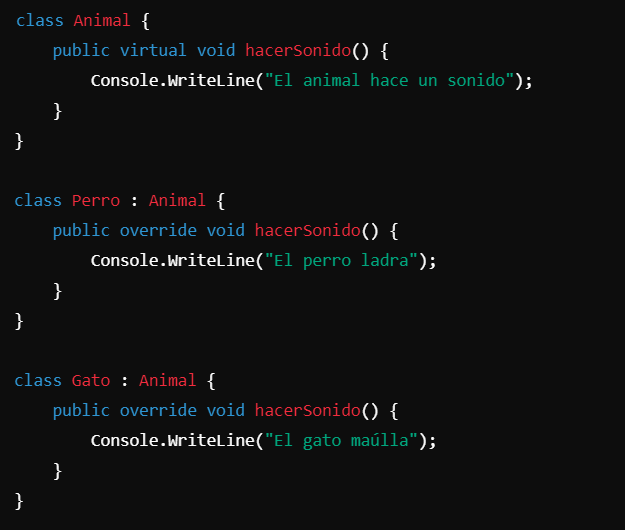
Description automatically generated

**Polimorfismo**

El **polimorfismo** permite que un mismo método pueda tener diferentes comportamientos dependiendo del objeto que lo invoque. Es una forma de sobreescribir o sobrecargar métodos para que se adapten a diferentes situaciones.

**Ejemplo:**

Supongamos que tienes una clase Animal con un método hacerSonido(). Luego, puedes tener clases como Perro y Gato que heredan de Animal y sobreescriben el método hacerSonido() para que el perro ladre y el gato maúlle.



**Interfaces**

Una **interfaz** es un contrato que define qué métodos debe tener una clase, pero no dice cómo se implementan. Es útil cuando diferentes clases deben cumplir con un conjunto de comportamientos, pero cada clase puede implementarlos de forma diferente.

**Ejemplo:**

Si tienes una interfaz IVehiculo que define un método mover(), cualquier clase que implemente esta interfaz deberá incluir el método mover(), aunque puede definirlo de maneras distintas.

A computer screen shot of text

Description automatically generated

**Abstracción**

La **abstracción** consiste en mostrar solo los detalles esenciales de un objeto y ocultar la implementación compleja. Esto permite que los programadores trabajen con objetos de manera más simple sin preocuparse por cómo están construidos internamente.

**Ejemplo:**

Un coche tiene muchas partes complejas como el motor, la transmisión, etc., pero para el usuario solo es importante saber cómo arrancarlo, acelerar o frenar. Esa es la abstracción: se ocultan los detalles complejos y se expone solo lo necesario.

A computer screen with white text

Description automatically generated

}

**Principios de la Programación Orientada a Objetos**

Existen cuatro principios básicos en la POO:

1. **Encapsulamiento**: Protege los datos dentro de un objeto para que solo se puedan modificar mediante métodos definidos, evitando el acceso directo.
2. **Abstracción**: Oculta los detalles innecesarios y muestra solo lo esencial.
3. **Herencia**: Permite crear nuevas clases a partir de otras existentes.
4. **Polimorfismo**: Permite que un mismo método tenga diferentes comportamientos según el contexto.

**Ejemplo:**

El **encapsulamiento** se puede ver cuando usamos propiedades privadas y las manipulamos solo a través de métodos públicos.

A computer screen shot of text

Description automatically generated

**Buenas practicas en la creacion de Interfaces**

**1. Nombres Claros y Significativos**

El nombre de una interfaz debe reflejar claramente su propósito. En C#, es común que los nombres de las interfaces empiecen con una "I" mayúscula para indicar que son interfaces.

**Ejemplo:**

A black screen with red and blue text

Description automatically generated

El nombre IEnviable indica que las clases que implementen esta interfaz tendrán un comportamiento relacionado con el envío.

**2. Seguir el Principio de Responsabilidad Única (SRP)**

Una interfaz debe tener una responsabilidad clara y única. No debe abarcar demasiados métodos o responsabilidades diferentes. Si es necesario, divide la interfaz en varias más pequeñas.

**Ejemplo:**

En lugar de tener una interfaz como esta:

A screen shot of a computer program

Description automatically generated

Es mejor dividirla en interfaces más específicas:

A screen shot of a computer code

Description automatically generated

**3. Seguir el Principio de Segregación de Interfaces (ISP)**

El **Principio de Segregación de Interfaces (ISP)** dicta que una clase no debe ser forzada a implementar métodos que no necesita. Crear interfaces pequeñas y específicas ayuda a cumplir con este principio.

**Ejemplo:**

En lugar de crear una interfaz IVehiculo que incluya métodos que no todos los vehículos necesitarán, como RepostarCombustible() para vehículos eléctricos, divide las interfaces:

A computer screen with text on it

Description automatically generated

De esta manera, solo los vehículos que necesitan repostar combustible implementarán la segunda interfaz.

**4. No Implementar Métodos en la Interfaz**

Una interfaz debe ser solo un contrato, no debe incluir implementación de métodos. La lógica de los métodos debe ser implementada por las clases que implementan la interfaz.

**Ejemplo:**

Esto es incorrecto:

A black screen with white text

Description automatically generated

En lugar de eso, la interfaz debe definir únicamente la firma del método:

A black screen with red and blue text

Description automatically generated

Y cada clase que implemente la interfaz proporciona su propia implementación:

A screen shot of a computer code

Description automatically generated

**5. Evitar Cambios Frecuentes en Interfaces**

Una vez que una interfaz se define y comienza a ser implementada por múltiples clases, cambiarla puede ser problemático y romper código que ya está funcionando. Por eso, es importante diseñarlas de manera sólida desde el principio.

**Solución:**

Si necesitas hacer cambios o agregar métodos, en lugar de modificar la interfaz original, puedes crear una nueva interfaz que amplíe la existente.

A computer screen with text on it

Description automatically generated

**6. Interfaces para Definir Comportamientos, no Detalles de Implementación**

Las interfaces deben enfocarse en **qué** hace una clase, no en **cómo** lo hace. No incluyas detalles de implementación en la interfaz.

**Ejemplo:**

En lugar de:

A black screen with red and blue text

Description automatically generated

Una mejor práctica es:

A black screen with white text

Description automatically generated

Esto permite a las clases implementar la conexión en distintos tipos de bases de datos, sin estar limitadas a MySQL.

**7. Uso de Interfaces para Soporte de Polimorfismo**

Utiliza interfaces cuando diferentes clases deben compartir el mismo comportamiento. Esto te permite aplicar **polimorfismo**, donde diferentes objetos pueden ser tratados de manera uniforme.

**Ejemplo:**

A screen shot of a computer program

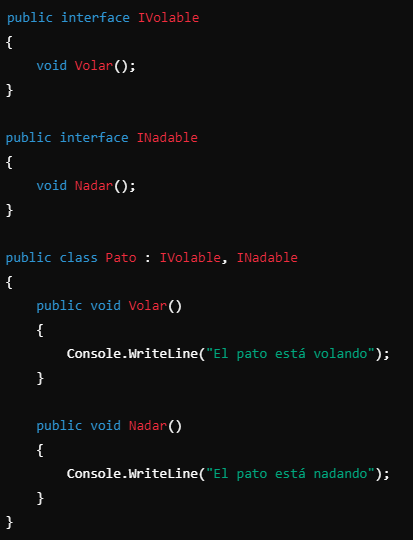
Description automatically generated

En este ejemplo, cualquier clase que implemente IMensaje puede ser pasada al método Notificar(), sin importar si es un Email o un SMS.

**8. Implementar Múltiples Interfaces Cuando Sea Necesario**

En C#, una clase puede implementar varias interfaces. Si una clase necesita cumplir con varios comportamientos, puedes hacer que implemente más de una interfaz.

**Ejemplo:**



**9. Prefiere Interfaces a Herencia Múltiple**

Como en C# no existe la herencia múltiple (una clase no puede heredar de más de una clase), es mejor usar interfaces si necesitas que una clase tenga comportamientos de múltiples tipos.

**Ejemplo:**

Si tienes una clase Vehiculo, no puedes hacer que herede tanto de Aereo como de Acuatico, pero puedes implementar ambas interfaces IVolable e INadable como en el ejemplo anterior.

**10. Apoyar Dependencia de Interfaces (DIP)**

El **Principio de Inversión de Dependencia (DIP)** dice que las clases deben depender de interfaces o abstracciones, no de clases concretas. Esto mejora la flexibilidad y testabilidad del código.

**Ejemplo:**

En lugar de hacer que una clase dependa de una implementación específica:

A computer screen shot of a program

Description automatically generated

Haz que dependa de una interfaz:

A computer screen shot of a program code

Description automatically generated

De esta manera, puedes pasar cualquier clase que implemente IMensaje (como Email o SMS).

**Cuándo Utilizar Interfaces**

Las **interfaces** se usan cuando necesitas definir un conjunto de comportamientos que diferentes clases pueden implementar, sin importar cómo lo hagan. Las interfaces solo definen **qué** se debe hacer, no **cómo** se hace. Son ideales cuando varias clases no están relacionadas jerárquicamente pero comparten comportamientos comunes.

**Usos principales:**

1. **Definir un contrato** que debe cumplirse sin importar la implementación.
2. Cuando necesitas **polimorfismo** sin depender de una jerarquía de clases.
3. Cuando una clase necesita implementar múltiples comportamientos que pueden provenir de diferentes fuentes (C# permite múltiples interfaces, pero no herencia múltiple).

**Ejemplo:**

A screenshot of a computer program

Description automatically generated

En este ejemplo, tanto Pato como Avion implementan el comportamiento de volar, pero no comparten una relación directa en su jerarquía. Las interfaces permiten que cualquier clase implemente los métodos Volar() y Nadar() sin estar forzadas a heredar de una clase común.

**Usa interfaces cuando:**

* Quieras forzar a las clases a que implementen un conjunto específico de métodos sin importar su jerarquía.
* Necesites que una clase implemente múltiples comportamientos sin herencia múltiple (ya que C# no la permite).
* Quieras que la implementación de los métodos sea responsabilidad de cada clase.

**Cuándo Utilizar Clases Abstractas**

Una **clase abstracta** se usa cuando tienes un comportamiento o atributos que quieres compartir entre varias clases, pero también quieres permitir que las clases hijas proporcionen su propia implementación de algunos métodos. Las clases abstractas **pueden** tener tanto métodos con implementación como métodos abstractos (sin implementación), que las clases derivadas deben sobrescribir.

**Usos principales:**

1. Cuando tienes una **relación jerárquica** clara entre clases.
2. Cuando necesitas compartir alguna implementación común pero también obligar a las clases derivadas a implementar ciertos comportamientos específicos.
3. Cuando quieres definir algunos **métodos abstractos** (sin implementación) que deben ser implementados por las clases derivadas.

**Ejemplo:**

A screen shot of a computer program

Description automatically generated

En este ejemplo, la clase Animal define un método concreto Respirar() que todas las clases hijas pueden usar, pero también tiene un método abstracto HacerSonido() que obliga a las clases derivadas (Perro y Gato) a proporcionar su propia implementación.

**Usa clases abstractas cuando:**

* Tienes un **comportamiento común** que todas las clases derivadas deben compartir, pero también necesitas que proporcionen sus propios detalles específicos.
* Estás definiendo una jerarquía donde las clases hijas tienen una relación **específica** con la clase base.
* Quieres tener tanto **métodos con implementación** como métodos sin implementación.

**Cuándo Utilizar Herencia**

La **herencia** es una técnica para reutilizar el código, donde una clase hereda comportamientos y características de otra clase. La herencia define una relación **"es un"** (e.g., un Coche es un Vehículo), y permite compartir funcionalidades comunes entre clases. A diferencia de las interfaces, solo puedes heredar de una única clase base en C# (no existe la herencia múltiple).

**Usos principales:**

1. **Compartir funcionalidad** común entre clases relacionadas jerárquicamente.
2. Cuando una clase base tiene atributos o métodos que pueden ser reutilizados por las clases derivadas.
3. Cuando las clases derivadas necesitan **extender o modificar** el comportamiento de la clase base.

**Ejemplo:**

A screen shot of a computer program

Description automatically generated

En este caso, Coche hereda de Vehiculo, lo que significa que puede utilizar los métodos Arrancar() y Detener() definidos en Vehiculo, pero también puede agregar comportamientos propios como TocarBocina().

**Usa herencia cuando:**

* Exista una **relación directa y jerárquica** entre las clases (e.g., un coche es un tipo de vehículo).
* Quieres reutilizar el código de una clase base en múltiples clases derivadas.
* La clase base tiene sentido por sí misma y sus atributos o métodos son aplicables a las clases derivadas.

**Comparación y Casos de Uso**

|  |  |  |  |
| --- | --- | --- | --- |
| Característica | Interfaces | Clases Abstractas | Herencia |
| Múltiples implementaciones | Se puede implementar múltiples interfaces en una clase. | No se puede heredar de múltiples clases abstractas. | No se permite herencia múltiple. |
| Métodos con implementación | No puede tener implementación de métodos. | Puede tener métodos con o sin implementación. | La clase base puede tener cualquier tipo de método. |
| Relación | Define comportamiento común para clases no necesariamente relacionadas. | Define comportamiento para clases relacionadas jerárquicamente. | Establece una relación padre-hijo clara. |
| Uso típico | Polimorfismo sin herencia, implementación múltiple de comportamientos. | Compartir comportamiento común y forzar a las clases hijas a implementar métodos específicos. | Reutilización de código y extensión de funcionalidad. |
| Jerarquía de clases | No establece jerarquía. | Establece una jerarquía parcial (puede ser base de otras clases). | Establece una relación jerárquica completa. |

**Resumiendo:**

* **Usa interfaces** cuando quieras definir un conjunto de métodos que deben ser implementados por clases no necesariamente relacionadas entre sí, y cuando quieras aprovechar el polimorfismo sin herencia múltiple.
* **Usa clases abstractas** cuando necesitas compartir tanto comportamiento común como contratos obligatorios entre clases que tienen una relación jerárquica clara.
* **Usa herencia** cuando tengas una jerarquía clara de clases con comportamientos compartidos que deben ser reutilizados o extendidos.

La clave está en elegir la técnica que mejor se ajuste a la relación entre las clases y los comportamientos que necesitas modelar.