INSTITUTO TECNOLOGICO DEL SUR DE NAYARIT

ALUMNO:

Javier Eduardo Ramirez Ornelas

MAESTRA:

Cinthia Anahí Mata Bravo

NUMERO DE CONTROL:

191140026

MATERIA:

PROGRAMACION ORIENTADA A OBJETOS

CARRERA:

ITIC

INDICE

1. ¿Qué es la interfaz de una clase?

En teoría de orientación a objetos, la interfaz de una clase es *todo lo que podemos hacer con ella*. A efectos prácticos: todos los métodos, propiedades y variables públicas (aunque no debería haber nunca variables públicas, debemos usar propiedades en su lugar) de la clase conforman *su interfaz*.
Dada la siguiente clase:

```
class Contenedor
{
   public int Quitar();
   public void Meter(int v);
   private bool Esta Repetido(int v);
}
```

Su interfaz está formada por los métodos *Quitar* y *Meter*. El método *Esta Repetido* **no** forma parte de la interfaz de dicha clase, ya que es privado.
En orientación a objetos decimos que la interfaz de una clase define **el comportamiento** de dicha clase, ya que define que podemos y que no podemos hacer con objetos de dicha clase: dado un objeto de la clase *Contenedor* yo puedo llamar al método *Quitar* y al método *Meter* pero no puedo llamar al método *Esta Repetido*.
Así pues: **toda clase tiene una interfaz** que define que podemos hacer con los objetos de dicha clase.

2. Interfaces idénticas no significa clases intercambiables

Fíjate en estas dos clases:

```
class Contenedor
{
    public int Quitar() { ... }
    public void Meter (int v) { ... }
}
class OtroContenedor
{
    public int Quitar() { ... }
    public void Meter (int v) { ... }
}
```

¿Qué puedes deducir de ellas? ¡Exacto! Su interfaz es la misma: con ambas clases podemos hacer lo mismo: llamar al método *Quitar* y al método *Meter*.
Ahora imagina que en cualquier otro sitio tienes un método definido tal y como sigue:

```
public void foo (Contenedor c)
{
    // Hacer cosas con c como p.ej:
    int i = c.Quitar();
    c.Meter(10);
}
```

El método recibe un Contenedor y opera con él. Ahora dado que las interfaces de *Contenedor* y *Otro* Contenedor son iguales, uno podría esperar que lo siguiente funcionase:

```
Otro Contenedor oc = new Otro Contenedor();
foo(oc);
```

Pero esto **no va a compilar**. ¿Por que? Pues, aunque nosotros somos capaces leyendo el código de comparar la interfaz de ambas clases, el compilador no puede hacer esto. Para el compilador *Contenedor* y *OtroContenedor* son **dos clases totalmente distintas sin ninguna relación**. Por lo tanto, un método que espera un *Contenedor* no puede aceptar un objeto de la clase *OtroContenedor*.
Quiero recalcar que el hecho de que el compilador no *compare las interfaces de las clases* no se debe a una imposibilidad técnica ni nada parecido: se debe a que no tiene sentido hacerlo.
¿Por qué? Pues simplemente porque las interfaces son idénticas por pura casualidad.

¿Supón que fuese legal llamar a foo con un objeto OtroContenedor, ok?
Entonces podría pasar lo siguiente:

1. Alguien **añade** un método público a la clase Contenedor.
2. Se modifica el método foo para que llame a dicho método nuevo. Eso es legal porque foo espera un *Contenedor* como parámetro
3. La llamada a foo(oc) donde oc es OtroContenedor… como debe comportarse ahora? *OtroContenedor* no tiene el método nuevo que se añadió a Contenedor!
   Así pues: dos clases con la misma interfaz **no tienen relación alguna entre ellas y por lo tanto no se pueden intercambiar.**

3. Implementación de interfaces

El ejemplo anterior ejemplifica un caso muy común: tener dos clases que hacen *lo mismo,* pero de diferente manera. P.ej. imagina que Contenedor está implementado usando un array en memoria y *OtroContenedor* está implementando usando, que sé yo, pongamos un fichero en disco. La funcionalidad (la interfaz) es la misma, lo que varía es **la implementación.** Es por ello que en programación orientada a objetos decimos que las interfaces son funcionalidades (o comportamientos) y las clases representen implementaciones.
Ahora bien, si dos clases representan **dos implementaciones distintas de la misma funcionalidad**, es muy enojante (y estúpido) que no las podamos intercambiar. Para que dicho intercambio sea posible C# (y en general cualquier lenguaje orientado a objetos) permite explicitar la interfaz, es decir **separar la declaración de la interfaz de su implementación** (de su clase). Para ello usamos la palabra clave *interface*:

```
interface Contenedor
{
    int Quitar();
    void Meter(int i);
}
```

Este código declara una interfaz IContenedor que declara los métodos Quitar *y* Meter. Fíjate que los métodos no se declaran como public (en una interfaz la visibilidad no tiene sentido, ya que todo es public) y que **no se implementan los métodos**.
Las interfaces son un concepto más teórico que real. No se pueden crear interfaces. El siguiente código NO compila:

```
IContenedor c = new IContenedor();
// Error: No se puede crear una interfaz!
```

Es lógico que NO podamos crear interfaces, ya que si se nos dejara, y luego hacemos c.Quitar()… que método se llamaría si el método Quitar() no está implementado?

Aquí es donde volvemos a las clases: podemos indicar explícitamente que una clase **implementa** una interfaz, es decir **proporciona implementación (código) a todos y cada uno de los métodos (y propiedades) declarados en la interfaz:**

```
class Contenedor : IContenedor
{
    public int Quitar() { ... }
    public void Meter(int i) { ... }
}
```

La clase Contenedor declara explícitamente que implementa la interfaz IContenedor. Así pues, la clase **debe proporcionar implementación para todos los métodos** de la interfaz. El siguiente código p.ej. no compila:

```
class Contenedor : IContenedor
{
    public void Meter(int i) { ... }
}
// Error: Y el método Quitar()???
```

Es por esto que en orientación a objetos decimos que las interfaces son *contratos*, porque si yo creo la clase la interfaz me obliga a implementar ciertos métodos y si yo uso la clase, la interfaz me dice que métodos puedo llamar.
Y ahora viene lo bueno: Si dos clases **implementan la misma interfaz** son intercambiables. Dicho de otro modo, en cualquier sitio donde se espere una instancia de la interfaz puede pasarse una instancia de cualquier clase que implemente dicha interfaz.

Podríamos declarar nuestro método foo anterior como sigue:

```
void foo(IContenedor c)
{
    // Cosas con c...
    c.Quitar();
    c.Meter(10);
}
```

Fíjate que la clave es que el parámetro de foo está declarado como *IContenedor*, no como *Contenedor* o *OtroContenedor*, con esto indicamos que el método foo() trabaja con cualquier objeto de cualquier clase que implemente IContenedor.
Y ahora, si supones que tanto Contenedor como OtroContenedor implementan la interfaz IContenedor el siguiente código es válido:

```
Contenedor c = new Contenedor();
foo(c);      // Ok. foo espera IContenedor y Contenedor implementa IContenedor
OtroContenedor oc = new OtroContenedor();
foo(oc); // Ok. foo espera IContenedor y OtroContenedor implementa IContenedor
// Incluso esto es válido:
IContenedor ic = new Contenedor();
IContenedor ic2 = new OtroContenedor();
```

4. ¿Cuándo usar interfaces?

En general siempre que tengas, o preveas que puedes tener más de una clase para hacer lo mismo: usa interfaces. Es mejor pecar de exceso que de defecto en este caso. No te preocupes por penalizaciones de rendimiento en tu aplicación porque no las hay. ´

No digo que **toda** clase deba implementar una interfaz obligatoriamente, muchas clases **internas** no lo implementarán, pero en el caso de las clases **públicas (visibles desde el exterior**) deberías pensarlo bien. Además, pensar en la interfaz antes que, en la clase en sí, es *pensar en lo que debe hacerse* en lugar de pensar *en cómo debe hacerse*. Usar interfaces permite a posteriori cambiar una clase por otra que implemente la misma interfaz y poder integrar la nueva clase de forma mucho más fácil (sólo debemos modificar donde instanciamos los objetos, pero el resto de código queda igual).

Ejemplo:

Segregación de interfaces

Imagina que tenemos un sistema que debe trabajar con varios vehículos, entre ellos aviones y coches, así que declaramos la siguiente interfaz:

```
interface IVehiculo
{
    void Acelerar(int kmh);
    void Frenar();
    void Girar(int angulos);
    void Despegar();
    void Aterrizar();
}
```

Luego implementamos la clase avión:

```
class Avion : IVehiculo
{
    public void Acelerar(int kmh) { ... }
    public void Frenar() { ... }
    public void Girar (int angulos) { ... }
    public void Despegar() { ... }
    public void Aterrizar() { ... }
}
```

Y luego vamos a por la clase coche… y aquí surge el problema:

```
class Coche : IVehiculo
{
    public void Acelerar(int kmh) { ... }
    public void Frenar() { ... }
    public void Girar (int angulos) { ... }
    public void Despegar() {throw new NotImplementedException("Coches no vuelan"); }
    public void Aterrizar(){throw new NotImplementedException("Coches no vuelan"); }
}
```

La interfaz IVehiculo tiene demasiados métodos y no define el comportamiento de todos los vehículos, dado que no todos los vehículos despegan y aterrizan. En este caso es mejor dividir la interfaz en dos:

```
interface IVehiculo
{
    void Acelerar(int kmh);
    void Frenar();
    void Girar (int angulos);
}

interface IVehiculoVolador : IVehiculo
{
    void Despegar();
    void Aterrizar();
}
```

Fíjate además que IVehiculoVolador deriva de *IVehiculo* (en orientación a objetos decimos que hay una relación de herencia entre IVehiculoVolador y IVehiculo), **eso significa que una clase que implemente IVehiculoVolador debe implementar también IVehiculo forzosamente**. Por lo tanto podemos afirmar que todos los vehículos voladores son también vehículos.
Ahora sí que la clase Coche puede implementar IVehiculo y la clase Avión puede implementar IVehiculoVolador (y por lo tanto también IVehiculo). Si un método foo() recibe un objeto IVehiculoVolador puede usar métodos tanto de IVehiculoVolador como de IVehiculo:

```
void foo (IVehiculoVolador vv)
{
   vv.Acelerar(10);     // Ok. Acelerar es de IVehiculo y IVehiculoVolador deriva de IVehiculo
   vv.Despegar();       // Ok. Despegar es de IVehiculoVolador
}
```

Al revés no! Si un método foo recibe un IVehiculo **no puede llamar a métodos de IVehiculoVolador**. Lógico: todos los vehículos voladores son vehículos pero al revés no… no todos los vehículos son vehículos voladores!
Siempre que haya segregación no tiene por qué haber herencia de interfaces. Imagina el caso de que además de vehículos debemos tratar con Armas de guerra. Tenemos otra interfaz:

```
interface IArmaDeGuerra
{
    void Apuntar();
    void Disparar();
}
```

Ahora podrían existir clases que implementen *IArmaDeGuerra* como p.ej. una torreta defensiva:

```
class TorretaDefensiva : IArmaDeGuerra
{
    public void Apuntar() { ... }
    public void Disparar() { ... }
}
```

¡Pero claro… también tenemos vehículos que pueden ser a la vez armas de guerra, p.ej. un tanque! ¿Qué hacemos? **Ningún problema: ¡una clase puede implementar más de una interfaz a la vez!** Para ello debe implementar todos los métodos de todas las interfaces:

```
class Tanque : IVehiculo, IArmaDeGuerra
{
    public void Acelerar(int kmh) { ... }
    public void Frenar() { ... }
    public void Girar (int angulos) { ... }
    public void Apuntar() { ... }
    public void Disparar() { ... }
}
```

Ahora si un método foo () recibe un IVehiculo le puedo pasar un Tanque y si otro método foo2 recibe un IArmaDeGuerra también le puedo pasar un Tanque. ¡O, dicho de otro modo, los tanques se comportan como vehículos y como armas de guerra a la vez!

Así pues, es importante segregar bien nuestras interfaces porque en caso contrario vamos a tener dificultades a la hora de implementarlos. La segregación de interfaces es uno **de los 5 principios SOLID** (concretamente la letra I: interface segregation principle).

Resumen:

a lo que yo entendí la interfaz es como cuando utilizábamos un public void
o un switch que sirve para llamar clase no le entendí mucho, pero me parece que es como cuando ponemos variables privadas y las mandamos llamar con los dos puntos y adentro de la interfaz van variables que le podemos poner a otras cosas como se muestra en el ejemplo:

```
interface IArmaDeGuerra
{
    void Apuntar();
    void Disparar();
}
```

Ahora podrían existir clases que implementen IArmaDeGuerra como p.ej. una torreta defensiva:

```
class TorretaDefensiva : IArmaDeGuerra
{
    public void Apuntar() { ... }
    public void Disparar() { ... }
}
```

para mi es para tener mejor estructura en tu programa es decir la interfaz cuenta con las variables apuntar y disparar
y ya en tu class aquí es torreta ofensiva y ya pones los dos puntos y mandas llamar la interfaz que tiene variables que le sirven a la clase torreta ofensiva
y ya nomás las agregas con un public void.
Y las utilizas.

Se utilizan:

En general siempre que tengas, o preveas que puedes tener más de una clase para hacer lo mismo. No digo que toda clase deba implementar una interfaz obligatoriamente, muchas clases internas no lo implementarán, pero en el caso de las clases públicas (visibles desde el exterior) deberías pensarlo bien.