Tema 3 – Encapsulamiento, herencia y polimorfismo

Herencia

Por medio de esta característica se pueden definir clases a partir de otras, más generales, y solo agregar las propiedades y métodos de la especialización. El sistema será una colección de clases relacionadas entre sí, por eso es importante el modo en que estará conformada su estructura.

Un diseño pobre genera más problemas que soluciones, ya que los errores se propagan rápidamente, y su expansión y mantenimiento se dificultan.

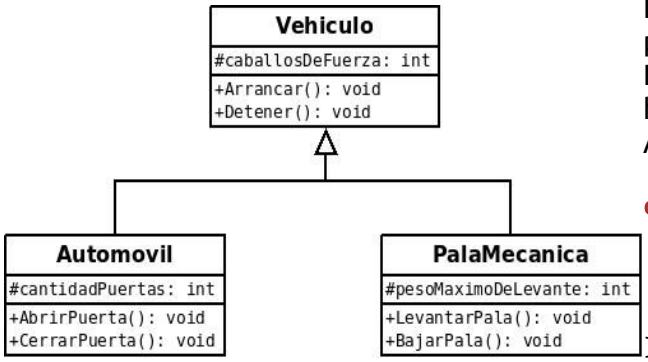
```
class A{
}

class B : A{
}
```

La clase B posee, seguido de su identificador, el carácter dos puntos(:) y, luego, el identificador de la clase de la cual deriva. **La clase B heredará los métodos y variables públicas, protegidas e internas de A.**

Hay métodos y variables que pueden ser comunes en varias clases, al usar herencia se evita tener que reescribirlos.

Por ejemplo se puede tener una clase Vehículo que tenga las variables y los métodos comunes a otras clases que hereden, luego estas clases hijas sólo deberán agregar los métodos específicos del tipo de vehículo que representan. En el caso de existir un error en un método de la clase padre, sólo habría que modificar el código de una clase y no de varias clases.

```
class Vehiculo{
	protected int caballosDeFuerza;

	public void Arrancar(){
		Console.WriteLine("Arranque");
	}
	public void Detener(){
		Console.WriteLine("Parada");
	}
}
class Automovil : Vehiculo{
	protected int cantidadPuertas;

	public void AbrirPuerta(int puertaNum){
		Console.WriteLine("Abrir puerta");
	}
	public void CerrarPuerta(int puertaNum){
		Console.WriteLine("Cerrar puerta");
	}
}
class PalaMecanica : Vehiculo{
	protected int pesoMaximoDeLevante;

	public void LevantarPala(){
		Console.WriteLine("Levantar Pala");
	}
	public void BajarPala(){
		Console.WriteLine("Bajar Pala");
	}
}
```



La relación entre las clases heredadas y la clase padre es una relación de generalización.
Las variables y métodos en este ejemplo son heredados por eso es posible invocar al método Arrancar desde la instancia de Automovil.

```
class MainClass{
 public static void Main(){
       Automovil unAuto = new Automovil();
       unAuto.Arrancar();
 }
}
```

En C# toda clase será subclase de la clase Objecto(aunque no se indique explícitamente). Toda clase de la librería BCL es subclase de Object, que se halla en el namespace System. Siempre los objetos podrán ser considerados del tipo Object, una ventaja que se puede llegar a explotar.

Redefinir métodos en clases heredadas

La clase Vehiculo posee el método Arrancar, que probablemente sirva para cualquier vehículo genérico. Si se define otro método Arrancar en la clase Automovil entonces existirían dos métodos Arrancar.

Lo común y lógico es que parte del proceso de encendido de un automóvil sea común al de cualquier vehículo.

```
class Vehiculo{
       public void Arrancar(){
              Console.WriteLine("Primera fase de arranque " +
                     "comun a todos los vehiculos");
       }
}

class Automovil : Vehiculo{

       //redifinicion de metodo para
       //adaptarlo a la especificacion de
       //la clase
       public new void Arrancar(){
              base.Arrancar();//llamada a metodo de la clase base
              Console.WriteLine("Parte de arranque" +
                           " especifica de los automoviles");
       }
}

class MainClass{
       public static void Main(){
              Automovil unAuto = new Automovil();
              unAuto.Arrancar();
       }
}
```

New también sirve para indicar la sobrescritura de métodos de la clase base, esto sirve para indicar que se sobrescribe un método de una clase padre. **Base** se utiliza cuando se desea acceder a métodos, variables o propiedades de la clase padre.

Uso de modificadores de acceso en herencia

Las variables declaradas como protected en la clase base no podrán ser accedidas desde fuera de la clases derivadas.

Modificador de acceso/ Accesible desde	Clase (donde se declaró)	Subclase (mismo assembly)	Subclase (distinto assembly)	Externamente (mismo assembly)	Externamente (distinto assembly)
private	si	no	no	no	no
internal	si	si	no	si	no
protected	si	si	si	no	no
Protected internal	si	si	si	si	no
public	si	si	si	si	si

¿Qué es un Assembly?

Un sistema siempre debe estar compuesto por uno o más assemblies. **Un assembly puede ser un archivo ejecutable o una librería de enlace dinámico, que puede poseer código ejecutable y/o recursos(bitmaps, iconos ocualquier tipo de dato).** Cada assembly posee un manifiesto que describe su contenido e informa de otras propiedades como su nombre, versión, etc.

Declarar variables como públicas no es una práctica aconsejable cuando:

- no todo el rango del tipo de dato de la variable debería ser aceptado
- de deba disparar algún evento antes de la asignación o después de ella
- la variable debe ser leída pero no escrita, o viceversa

Declaración de propiedades (get y set)

Las propiedades brindan la posibilidad de disparar código relacionado con la lectura o escritura de una variable sin perder el estilo de codificación.

```
class Vehiculo{
	protected int kilometros;
	public int Kilometros{
		get{
			return kilometros;
		}
		set{
			kilometros = value;
		}
	}
}
class MainClass{
	public static void Main(){
		Vehiculo unAuto = new Vehiculo();
		unAuto.Kilometros = 20;
		Console.WriteLine(unAuto.Kilometros);
	}
}
```

Una propiedad posee un modificador de acceso y un tipo de dato como si fuese una variable o un método convencional, pero inmediatamente después de su identificador posee un bloque declarativo dentro del cual existe un get (leer) y un set (fijar), que, a su vez, poseen bloques donde de coloca código.

Invocar constructores en herencia

```
class A{
	public A(){
		Console.WriteLine("Constructor de A");
	}
}

class B : A{
	public B(){
		Console.WriteLine("Constructor de B");
	}
}

class C : B{
	public C(){
		Console.WriteLine("Constructor de C");
	}
	static void Main(){
		C obj = new C();
	}
}
```

En este ejemplo al crear el objeto de la Clase C, que deriva de A y B, se invocarán los constructores de A y B aparte del de C.

Constructor de A
Constructor de B
Constructor de C

Esto es útil para tareas de inicialización en programas donde al crear un objeto se inicialicen tareas como abrir ficheros o canales de red etc). Pero si se desea invocar en alguna clase base un constructor diferente hay que hacer uso de la palabra reservada **base**:

```
class B{
	public B(){
		Console.WriteLine("Constructor de B");
	}
	public B(int num){
		Console.WriteLine("Constructor de B con parametro {0}", num);
	}
}
class C : B{
	public C(): base(10){
		Console.WriteLine("Constructor de C");
	}
	static void Main(){
		C obj = new C();
	}
}
```

Invocar destructores en herencia

Los destructores son invocados por el GC en orden inverso al que siguen los constructores.

```
class A{
	public A(){
		Console.WriteLine("Constructor de A");
	}
	~A(){
		Console.WriteLine("Destructor de A");
	}
}
class B : A{
	public B(){
		Console.WriteLine("Constructor de B");
	}
	~B(){
		Console.WriteLine("Destructor de B");
	}
}
class C : B{
	public C(){
		Console.WriteLine("Constructor de C");
	}
	~C(){
		Console.WriteLine("Destructor de C");
	}
	static void Main(){
		C obj = new C();
	}
}
```

Es un proceso inverso al de inicialización:

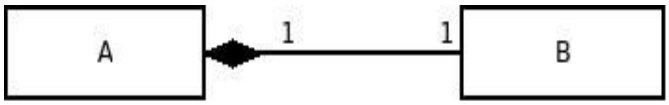
Constructor de A
Constructor de B
Constructor de C
Destructor de C
Destructor de B
Destructor de A

Composición

En un sistema será bastante frecuente que los objetos se encuentren compuestos por otros objetos.

```
class A{
	public void m1(){
		Console.WriteLine("Metodo de clase A");
	}
}
class B{
	public A objA = new A();
}
```

Todo objeto de la clase B se encontrará conformado por un objeto de la clase A.



El anterior ejemplo se representaría en un diagrama por medio de una relación de composición

A través de la utilización de un objeto de la clase B, se puede tener acceso al objeto A por medio de la variable objA.

```
class MainClass{
      public static void Main(){
            B objB = new B();
            objB.objA.m1();
      }
}
```

Composición frente a herencia

Se utiliza herencia cuando se desea incorporar a una clase las variables, propiedades y métodos del objeto seleccionado o cuando se desee especializar una clase agregando características de una clase base del mismo tipo, pero más general. La composición se utiliza para ocultar y encapsular el objeto seleccionado bajo una clase.

POLIMORFISMO

El polimorfismo es la habilidad que poseen los objetos para reaccionar de modo diferente ante los mismos mensajes. Por ejemplo un objeto de tipo Puerta, al igual que un objeto de tipo Ventana, podrá recibir el mensaje Abrir; sin embargo, cada uno de ellos reaccionará de modo diferente. El polimorfismo está relacionado con el mecanismo de sobrecarga y con los métodos virtuales. Las clases abstractas y las interfaces participan del polimorfismo ya que permiten tener varios métodos que posean el mismo nombre pero que realicen diferentes actividades en cada clase.

Sobrecarga de métodos

Es posible crear métodos de clases que posean el mismo nombre y difieran en la cantidad y/o en el tipo de parámetros.

Los métodos sobrecargados tampoco podrán diferenciarse exclusivamente en el orden de los parámetros (si es que éstos son del mismo tipo).

```
static void Imprimir(int num);
static void Imprimir(char car);
static void Imprimir(float num);
```

- **C# no soporta la declaración de parámetros opcionales.**
- No es posible crear dos métodos con el mismo identificador que sólo difieran en el tipo de dato devuelto.
- No es posible que dos métodos sobrecargados sólo difieren en el modificador static.

Sobrecarga de operadores

C# permite sobrecargar operadores, redefiniendo el significado de un conjunto de operadores cuando se utilizan con ciertos objetos.

```
static public Vector2 operator+(Vector2 vec1, Vector2 vec2){
	return new Vector2(vec1.X + vec2.X, vec1.Y + vec1.Y);
}
```

La declaración del operador es muy similar a la de un método, sólo que el identificador de éste se encuentra formado por la palabra **operator** y el operador que se desea sobrecargar.

El método que define la sobrecarga de un operador debe ser público y estático.

No todos los operadores pueden ser sobrecargados.

```
class Vector2{
	public float X = 0.0f;
	public float Y = 0.0f;

	Vector2(float _x, float _y){
		this.X = _x;
		this.Y = _y;
	}

	static public Vector2 operator+(Vector2 vec1, Vector2 vec2){
		return new Vector2(vec1.X + vec2.X, vec1.Y + vec2.Y);
	}

	public static void Main(){
		Vector2 vec1 = new Vector2(5.0f, 5.0f);
		Vector2 vec2 = new Vector2(2.0f, 2.0f);
		Vector2 vec3 = vec1 + vec2; // se invoca el metodo operador de suma
		Console.WriteLine("X {0}, Y {1}", vec3.X, vec3.Y);
	}
}
```

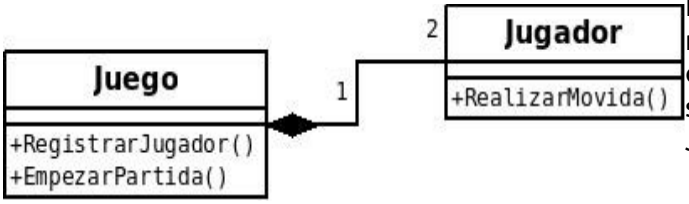
Métodos virtuales

La motivación principal de la creación de este tipo de métodos es la de poder separar la interfaz de una entidad con su implementación específica.

Ejemplo paso a paso explicativo:

Juego de ajedrez en donde se desea crear distintas inteligencias que tengan diferente implementación, pero que compartan una misma interfaz.

```
class Jugador{
	public void RealizarMovida(){
		Console.WriteLine("Clase: Jugador, Metodo: RealizarMovida");
	}
}
class Juego{
	Jugador jugador1;//objeto que representa al primer jugador
	Jugador jugador2;//objeto que representa al segundo jugador
	public void EmpezarPartida(Jugador jug1, Jugador jug2){
		this.jugador1 = jug1;
		this.jugador2 = jug2;
		jugador1.RealizarMovida();
		jugador2.RealizarMovida();

	}
}
```



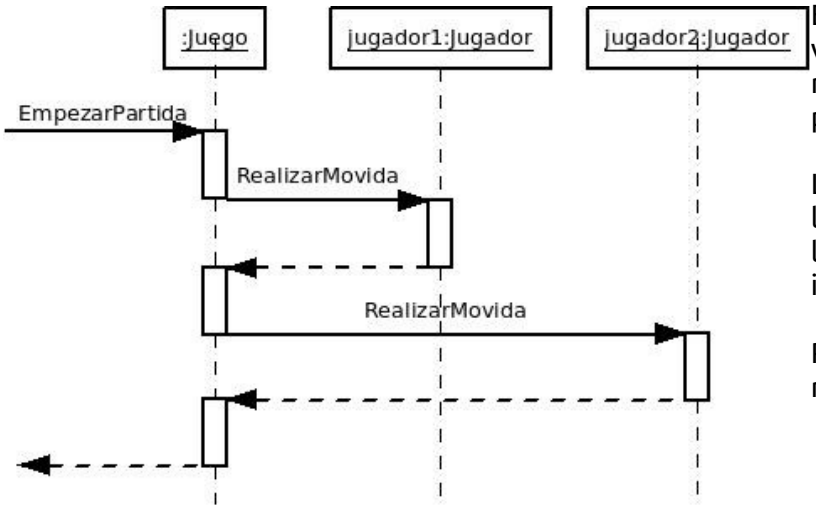
La relación de composición quedaría de esta manera. En donde un Juego puede tener 2 objetos de la clase Jugador y un Jugador solo puede estar en una instancia de la clase Juego.

El método Main sería el siguiente:

```
class MainClass{
	public static void Main(){
		Jugador jug1 = new Jugador();
		Jugador jug2 = new Jugador();
		Juego j = new Juego();
		j.EmpezarPartida(jug1, jug2);
	}
}
```

En pantalla la salida sería:

Clase: Jugador, Metodo: RealizarMovida
Clase: Jugador, Metodo: RealizarMovida



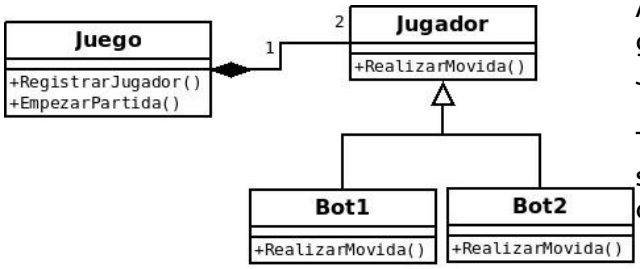
El diagrama de secuencias permite ver de forma gráfica cómo se deben realizar las llamadas a los métodos por parte de los objetos.

Los objetos son representados por los rectángulos superiores y las líneas horizontales representan la invocación de métodos.

Remitirse al módulo UML para repasar conceptos.

Para implementar las distintas inteligencias se van a crear subclases de Jugador que implementarán el comportamiento de cada tipo; podrían ser Bot1 y Bot2.

```
class Bot1 : Jugador{
	public void RealizarMovida(){
		Console.WriteLine("Clase: Bot1, Metodo: RealizarMovida");
	}
}

class Bot2 : Jugador{
	public void RealizarMovida(){
		Console.WriteLine("Clase: Bot2, Metodo: RealizarMovida");
	}
}
```



Ahora el diagrama de clases hay una generalización entre las clases especializadas y Jugador.

Tanto Bot1 como Bot2 se pretende que sobrecargar el método RealizarMovida de la clase base.

Se modifica la clase MainClass

```
class MainClass{
	public static void Main(){
		Jugador jug1 = new Bot1();
		Jugador jug2 = new Bot2();
		Juego j = new Juego();
		j.EmpezarPartida(jug1, jug2);
	}
}
```

Al ejecutar la instancia de la clase Juego no invoca a los métodos de los objetos del tipo Bot1 y Bot2:

Clase: Jugador, Metodo: RealizarMovida
Clase: Jugador, Metodo: RealizarMovida

El métodos EmpezarPartida invocó el método RealizarMovida de un objeto, que si bien es del tipo Bot1, fue asignado a un parámetro del tipo Jugador; por lo tanto, cuando se realiza la invocación, se ejecuta la versión de Jugador.

Para modificar este comportamiento, hay que **declarar al método de la clase base RealizarMovida como virtual. Y para sobrecargar en los métodos de las clases especificas hay que agregar el modificador llamado override.**

```
class Jugador{
	public virtual void RealizarMovida(){
		Console.WriteLine("Clase: Jugador, Metodo: RealizarMovida");
	}
}
class Bot1 : Jugador{
	public override void RealizarMovida(){
		Console.WriteLine("Clase: Bot1, Metodo: RealizarMovida");
	}
}
class Bot2 : Jugador{
	public override void RealizarMovida(){
		Console.WriteLine("Clase: Bot2, Metodo: RealizarMovida");
	}
}
```

Ahora la clase Juego está invocando un método de una clase (Bot1, Bot2) que ni siquiera conoce directamente. Esta es una característica de polimorfismo que brindan los métodos virtuales.

Se podrían crear nuevas subclases de Jugador y hacer que Juego las invoque sin modificar una sola línea de esta última clase.

Interfaces

Un interface es un "contrato de implementación" entre clases, donde se agrupan las clases por lo "que hacen", a diferencia de la herencia que agrupa las clases por "lo que son".

Por ejemplo, en la herencia se puede tener una clase padre que se llame "Persona" y existirán clases hijas (por ej. Cliente, Vendedor, etc) que heredarán sus características manteniéndose una relación de parentesco.

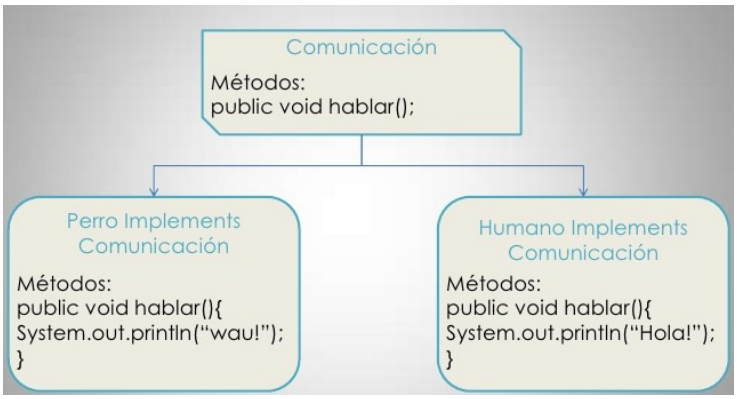
Para el caso de las interfaces se agrupan clases que hacen lo mismo, y no tienen por qué mantener una relación de parentesco.

Por ejemplo, se puede tener una interface que se llame "IEnumerable" que declare un método "Enumerar". Las clases que implementen la interface tienen la obligación de implementar el método "Enumerar".

Se pueden agrupar clases que hagan lo mismo, por ejemplo: Equipo, Carrito, Calendario etc. En todas las clases que implementen la interface implementarán el método Enumerar, que permitirá enumerar los elementos de esas clases. No hay una relación directa de parentesco entre las clases pero se agrupan por una funcionalidad que se requiere que hagan, que es enumerar.

Esto sirve para el polimorfismo apoyándose en la interface, en donde crear un método que muestre elementos de cualquier objeto que implemente la interfaz "IEnumerable".

En resumen las interfaces son declaraciones de métodos que objetos en común pueden compartir, inclusive si esos objetos no guardan relación ninguna. Si hay una serie de objetos que mediante la herencia no pueden conectarse o no pueden heredar métodos de un objeto padre ya que no tendría sentido, aquí es donde la interfaz juega un papel muy importante. Aparte las interfaces sirven para brindar a clases, sin importar cuales sean, de cierta funcionalidad que tienen que implementar obligadamente. Un ejemplo ilustrado:



- Una interface solo posee declaraciones
- Una interface puede declarar propiedades(declarando el get o set pero sin implementación.
- Una interface NO puede poseer variables ni constantes
- Una interface NO puede heredar de ninguna clase, pero SÍ de otras interfaces.
- Una clase que herede de una interface debe implementar TODOS los métodos declarados por ella.

- Todos los métodos declarados dentro de un interface siempre son públicos por defecto.
- Todos los métodos en una interface se comportan como virtuales.
- No se pueden crear instancias de un interface

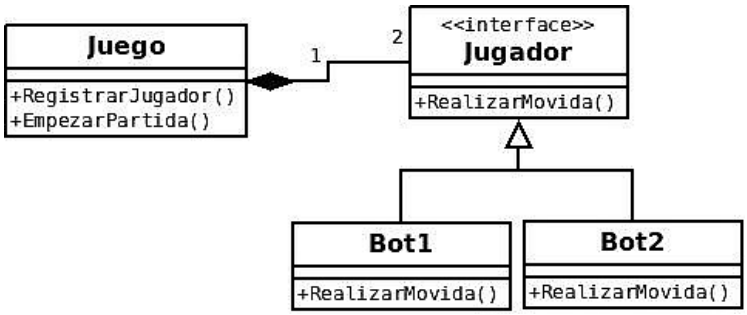
Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas. Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface.

```
using System;

namespace ejemplos.Interface
{
	interface ITest{
	  string HolaMundo();
	  string DevolverCadena(string s);
	}

	public class Minusculas : ITest{
		public string HolaMundo(){
			return ("hola, mundo");
		}

		public string DevolverCadena(string s){
			return s.ToLower();
		}
	}

	public class Mayusculas : ITest{
		public string HolaMundo(){
			return ("HOLA, MUNDO");
		}

		public string DevolverCadena(string s){
			return s.ToUpper();
		}
	}

	public class App{
		public static void Main(){
			// se declaran referencias a la interface y
			// se asocia una instancia de cualquiera
			// de las clases que la implementen
			ITest minusculas = new Minusculas();
			ITest mayusculas = new Mayusculas();

			Console.WriteLine(minusculas.HolaMundo());
			Console.WriteLine(mayusculas.HolaMundo());
			Console.WriteLine("---------------------------------------");

			Console.WriteLine(minusculas.DevolverCadena("Prueba de INTERFACES"));
			Console.WriteLine(mayusculas.DevolverCadena("Prueba de INTERFACES"));
		}
	}
}
```

La interface ITest conoce los métodos que tiene, pero no cómo ejecutarlos. Es por ello que, al asociarle una instancia que implemente la interface, la referencia se comportará de una forma u otra(polimorfismo). La utilización de interfaces es un recurso muy útil a la hora de crear, por ejemplo, conexiones a bases de datos.

Para el ejemplo del jugador anterior se podría establecer que todas las implementaciones de algún jugador debieran ser subclases de Jugador, y de esta clase base solo dejar el esqueleto(solo declaraciones de los métodos), ya que no cabe la posibilidad de que exista una instancia de Jugador, sino que existirían instancias de sus subclases.

```
interface iJugador{
	void RealizarMovida();
}
class Bot1 : iJugador{
	public void RealizarMovida(){
		Console.WriteLine("Clase: Bot1, Metodo: RealizarMovida");
	}
}
class Bot2 : iJugador{
	public void RealizarMovida(){
		Console.WriteLine("Clase: Bot2, Metodo: RealizarMovida");
	}
}
```



Clases abstractas

Las clases abstractas poseen la característica particular de **no poder ser instanciadas**. Éstas son creadas con el propósito de ser utilizadas posteriormente por medio de la herencia. En una clase abstracta se tiene una funcionalidad comun que se hereda para dar una funcionalidad concreta.

- Pueden definir métodos o declararlos como abstractos, para luego ser implementados por subclases.
- Las subclases declaradas también como abstractas no están obligadas a implementar los métodos abstractos de la clase base.
- Pueden poseer variables y constantes.
- Los métodos abstractos deben ser públicos.

```
abstract class Jugador{
	abstract public void RealizarMovida();
}
class Bot1 : Jugador{
	public override void RealizarMovida(){
		Console.WriteLine("Clase: Bot1, Metodo: RealizarMovida");
	}
}
class Bot2 : Jugador{
	public override void RealizarMovida(){
		Console.WriteLine("Clase: Bot2, Metodo: RealizarMovida");
	}
}
```

Se coloca la palabra reservada abstract antes de class y antes de la declaración del método que se comportará como un método virtual convecional (por lo que se coloca override antes de sobrecargarlo en subclases)

Diferencia entre clase abstracta e interface.

Aparte de las diferencias obvias entre una clase abstracta y un interface hay una diferencia que va mucho más allá entre estos dos conceptos. Mediante la herencia simple (clases base, abstractas, subclases) no se pueden agrupar clases dispares, en cambio con interfaces si se puede, es por esto que los interfaces proporcionan más polimorfismo que el que se puede obtener con una simple jerarquía de clases.

Ejemplo con Herencia simple:

```
abstract class Animal {
	public abstract void Habla();
}

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

class Gato : Animal{
	public override void Habla(){
		Console.WriteLine("Miau!");
	}
}

public class App{
	public static void Main(){
		Gato gato=new Gato();
		HazleHablar(gato);
	}
	static void HazleHablar(Animal sujeto){
		sujeto.Habla();
	}
}
```

El compilador no sabe exactamente que objeto se le pasará a la función HazleHablar en el momento de la ejecución del programa. Si se pasa un objeto de la clase Gato se imprimirá ¡Miau!, si se pasa un objeto de la clase Perro se imprimirá ¡Guau!. **El compilador solamente sabe que se le pasará un objeto de alguna clase derivada de Animal**. Por tanto, el compilador no sabe qué función Habla() será llamada en el momento de la ejecución del programa.

El polimorfismo ayuda a hacer el programa más flexible, ya que en el futuro se podrán añadir nuevas clases derivadas de Animal, sin que cambie para nada el método HazleHablar.

Ejemplo con interfaces:

```
interface IParlanchin {
	void Habla();
}
```

```
abstract class Animal : IParlanchin {
      public abstract void Habla();
}

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

class Gato : Animal{
      public override void Habla(){
            Console.WriteLine("Miau!");
      }
}

abstract class Reloj {

}

class Cucu : Reloj, IParlanchin{
      public void Habla(){
            Console.WriteLine("Cucu..Cucu!");
      }
}

public class App{
      public static void Main(){
            Gato gato=new Gato();
            HazleHablar(gato);
            Cucu cucu=new Cucu();
            HazleHablar(cucu);
      }
      static void HazleHablar(IParlanchin sujeto){
            sujeto.Habla();
      }
}
```

Al ejecutar el programa, se imprime en la consola ¡Miau!, por que a la función HazleHablar se le pasa un objeto de la clase Gato, y después ¡Cucu, cucu! por que a la función HazleHablar se le pasa un objeto de la clase Cucu.

Si solamente hubiese herencia simple, Cucu tendría que derivar de la clase Animal (lo que no es lógico) o bien no se podría pasar a la función HazleHablar. Con interfaces, cualquier clase en cualquier familia puede implementar el interface IParlanchin, y se podrá pasar un objeto de dicha clase a la función HazleHablar. Esta es la razón por la cual los interfaces proporcionan más polimorfismo que el que se puede obtener de una simple jerarquía de clases.

Actividades

1.- ¿En qué consiste la composición y para qué se usa?

2.- Una variable privada puede ser accedida desde (puede existir cero, una o varias opciones correctas):

a) La clase desde la cual fue creada.
b) Una subclase de la clase desde la cual fue declarada.
c) Externamente, desde un objeto de la clase la cual fue declarada.

3.- ¿Cuándo es conveniente regular el acceso a una variable por medio de una propiedad?

4.- ¿Para qué se utilizan los métodos virtuales?

5.- ¿Cuál es la diferencia fundamental entre una interface y una clase abstracta?

6.- ¿Puede una interface heredar de una clase?