SEMINARIO 7 C#

Alejandro E. Domínguez, Juan Carlos Vázquez, Juan Carlos Esquivel, Yandy Sánchez, Eric Martin

GRUPO: 2 Equipo 7

Herencia en C#:

- Única:
  Esto quiere decir que una clase puede heredar de una y solo una clase base de manera explícita

- Transitiva:
  De manera transitiva si **B** hereda de **A** (**B → A**) y **C** hereda de **B** (**C → B**) entonces todos los miembros del tipo **A** estarán disponibles en **B** y **C**

- Los constructores no son heredados (La clase derivada está obligada a llamar al constructor de la clase padre mediante el modificador *base*)

- Las clases derivadas pueden ocultar miembros heredados mediante el modificador (*new*), lo cual hace inaccesible el miembro heredado (esto no lo borra); y de manera análoga se pueden sobreescribir o invalidar miembros heredados marcando los mismos *(virtual* o *abstract*), esta última como su nombre indica solo deja el miembro como una declaración sin brindar implementación (abstracta)

- La accesibilidad de los miembros en una clase base, mediante los modificadores (*private*, *protected*, *internal*, *public*), afecta la visibilidad de los mismos en las clases derivadas

Herencia Implícita:

.NET tiene un sistema jerárquico de clases mediante un sistema de tipos, cualquier objeto hereda de manera implícita de object, garantizando una funcionalidad y raíz común para todos los tipos.

Interfaces:

Una `interface` maneja la forma de definir un tipo en su forma abstracta, mediante las funcionalidades que este debe tener. Dichas funcionalidades deberán luego ser implementadas por cada tipo que la contenga. Esto posibilita la implementación de más de una interfaz, lo que permite en ciertos casos simular herencia múltiple.

Polimorfismo:

A través de la herencia, una clase puede utilizarse como más de un tipo; puede utilizarse como su propio tipo, cualquier tipo base o cualquier tipo de interfaz si implementa interfaces. Esto se denomina polimorfismo. En C#, todos los tipos son polimórficos.

Conceptos de Varianza, Covarianza y Contravarianza:

Varianza: El concepto de **varianza** está relacionado con determinar en qué situaciones pueden usarse clases, interfaces, métodos y delegados definidos sobre un tipo **T** en lugar de las correspondientes clases, interfaces, métodos y delegados definidos sobre un subtipo o un supertipo de **T**. Las capacidades de genericidad, que enriquecen el sistema de tipos y mejoran el enfoque de control estático de tipos, hacen ver con más interés este fenómeno de las varianzas.

Covarianza: La **covarianza** es la propiedad por la cual podemos utilizar instancias de clases más pequeñas como si fueran una instancia de una clase mayor. Este concepto no existe en ningún enumerable en C# 2.0 y 3.0, a excepción de los **arrays**, debido a que permite una brecha importante. Usemos de ejemplo la clase `Persona`, de la que heredan en un principio `Estudiante` y `Trabajador`:

```
Persona[] array_personas = new Estudiante[10];
array_personas[0] = new Estudiante("José");
array_personas[1] = new Trabajador("Pedro");
```

Esta operación no muestra ningún error en tiempo de compilación, sino en tiempo de ejecución (**Excepción no controlada: System.ArrayTypeMismatchException: Intento de obtener acceso a un elemento como un tipo incompatible con la matriz.**) ya que estamos insertando un elemento de tipo `Trabajador` dentro de un array que se supone fuera tipo `Estudiante`. Otro ejemplo, que sí nos da error en compilación sería:

```
List<Persona> list_personas = new List<Estudiante>();
```

Precisamente para que no podamos insertar elementos de tipo `Trabajador` en una lista de tipo `Estudiante`. Este error es bastante restrictivo, sobre todo cuando solamente queremos utilizar la lista de estudiantes en operaciones generales válidas para toda persona. Por esto C# 4.0 nos permite realizar operaciones covariantes **si y sólo si** se utiliza en contextos de salida. Un ejemplo sería:

```
IEnumerable<Persona> ienum_pers = new List<Estudiante> {new Estudiante("Pepe")};
```

Ya que por definición de la interfaz `IEnumerable`, su parámetro genérico sólo se utiliza en contextos de salida, lo cual es especificado al compilador por la palabra clave `out`.

```
public interface IEnumerable<out T> : IEnumerable
```

Al estar definido el delegado `Func()` con la especificación `out` en C# 4.0, la covarianza en C# se extiende a los propios tipos genéricos también:

```
Func<Persona> func_persona = () => new Estudiante("Roberto");

IEnumerable<Func<Persona>> ienum_func_pers = new List<Func<Estudiante>>();
```

Contravarianza: La contravarianza es la propiedad que nos permite utilizar instancias de clases más genéricas, como si fueran instancias de clases más específicas. Al igual que la covarianza, este concepto no existe en C# 3.0, por lo que este código daría error de compilación:

```
IComparer<Persona> comp_personas = new Comparador();
// Comparador es una clase que implementa IComparer, recibe dos personas y
//las compara de acuerdo al nombre

list_estudiantes.Sort(comp_personas);
// Donde list_estudiantes es de tipo List<Estudiante>
```

Lo que podría parecer intuitivo, ya que le estamos asignando un comparador de personas (concepto más general) a un parámetro que espera un comparador de estudiantes (concepto más restringido). Pero en realidad es todo lo contrario, ya que, si `Comparador` es capaz de comparar personas, entonces debería ser capaz de comparar estudiantes, ya que los estudiantes son personas. Ya que `Comparador` solo recibe valores para comparar, no habría ningún riesgo en permitir este tipo de situaciones. Esto es resuelto en C# 4.0 con al aplicar el concepto de contravarianza. En C# 4.0, la interfaz `IComparer` está definida de la siguiente forma:

```
public interface IComparer<in T>
```

La palabra clave `in` indica al compilador que el parámetro genérico `T` se va a utilizar únicamente en contextos de entrada. Gracias a esto es que es posible ejecutar el código anterior en C# 4.0. Este concepto también es permitido al trabajar con delegados:

```
delegate int Compara<in T>(T x, T y);
```

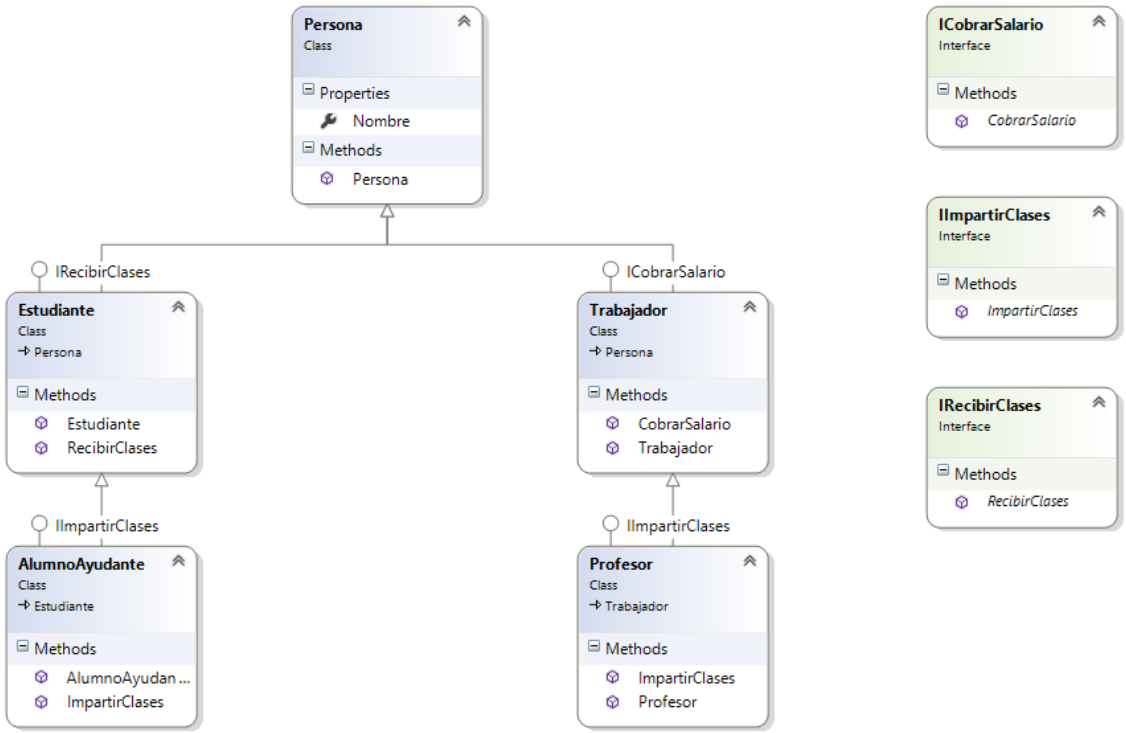
Ejercicio del Seminario:

En la Universidad, una persona (que se identifica por su Nombre) puede representar diferentes roles:

• Estudiante (Acción: **RecibirClase**())
• Profesor (Acción: **ImpartirClase**())
• Alumno Ayudante (Estudiante que no es profesor pero actúa como tal en un momento dado, es decir, puede realizar **ImpartirClase**())
• Trabajador (no todo trabajador es profesor, pero sí todos los profesores son trabajadores. Acción: **CobrarSalario**())

Inciso a: Diseñe una jerarquía en C# que represente/modele los roles anteriores y sus relaciones. Utilice alguna herramienta para ilustrar dicho modelo (Ejemplo: diseñador de clases de *Visual Studio*)

Respuesta:



Inciso b: ¿Es posible utilizar el siguiente código para imprimir una lista genérica de profesores? Haga los arreglos que crea necesario para que ejecute en caso de que su respuesta sea **NO**. Explique el funcionamiento de las características del lenguaje utilizadas.

```
void PrintPeople(IEnumerable<Persona> people) {
        for(var p in people)
                Console.WriteLine(p.Nombre);
}
```

Respuesta:

NO

Ejemplos de códigos validos:

```
public static void PrintPeople_Foreach(IEnumerable<Persona> people)
{
        foreach (var p in people)
        {
                Console.WriteLine(p.Nombre);
        }
}

public static void PrintPeople_For(IEnumerable<Persona> people)
{
        for (int i = 0; i < people.Count(); i++)
        {
                Console.WriteLine(people.ElementAt(i).Nombre);
        }
}
```

Inciso c: En la secretaría de la Facultad, usualmente se imprimen listados de estudiantes dado algún criterio (por nombre, por nota, etc). El algoritmo es el siguiente:

```
static void PrintStudents(IEnumerable<Estudiante> students, IComparer<Estudiante>
comparer)
{
        foreach (var student in students.OrderBy(x => x, comparer))
                Console.WriteLine(student.Nombre);
}
```

Implemente un comparador que permita utilizar el código anterior para imprimir los estudiantes ordenados por nombre, pero que dicho comparador se pueda reutilizar luego para profesores, trabajadores y alumnos ayudantes. Explique las características del lenguaje utilizadas.

Respuesta:

```
public class Comparador : IComparer<Persona>
{
	public int Compare(Persona x, Persona y)
	{
		return x.Nombre.CompareTo(y.Nombre);
	}
}


public static void PrintStudents(IEnumerable<Estudiante> students,
IComparer<Estudiante> comparer)
{
	foreach (var student in students.OrderBy(x => x, comparer))
		Console.WriteLine(student.Nombre);
}

public static void PrintPersonas(IEnumerable<Persona> people,
IComparer<Persona> comparer)
{
	foreach (var person in people.OrderBy(x => x, comparer))
		Console.WriteLine(person.Nombre);
}


Inciso_c.PrintStudents(list_estudiantes, new Comparador());

// Donde list_estudiantes es de tipo List<Estudiante>


Inciso_c.PrintPersonas(list_personas, new Comparador());

// Donde list_personas es de tipo List<Persona>
```

Esto es permitido por el concepto de contravarianza visto anteriormente. En el proyecto puede revisar que list_personas es un conjunto arbitrario de elementos de esta propia clases y clases que herdan de ella.

Inciso d: Explique e ilustre el funcionamiento del siguiente código:

```
static void PrintByConsole(Action<Action<Persona>> person)
{
        person(x => Console.WriteLine(x.Nombre));
}
...
PrintByConsole(x=>new Estudiante() {Nombre = "Pedro"});
```

Respuesta:

Primeramente, modificamos la última línea del código para que se adecuara al constructor de nuestra clase estudiante, que recibe un parámetro de tipo string.

```
Inciso_d.PrintByConsole(x => x(new Estudiante("Pedro"))
```

Esto no afecta al ejercicio, ya que suponemos que la razón por la que estaba especificado de esa forma es para garantizar que la propiedad Nombre existe en la clase Persona y sea heredada por Estudiante, para poder ser llamada luego en el método PrintByConsole.

Lo que causa duda en este ejercicio es que estamos usando el delegado Action (que utiliza el concepto de contravarianza) para una operación que parece en algún sentido covariante, es decir, estamos ingresando una instancia de la clase Estudiante en un método que recibe como parámetros un elemento de tipo Persona. Veamos por que sucede esto:

Otra forma de ver los conceptos de covarianza y contravarianza es la siguiente: la varianza “preserva la dirección de la herencia” mientras que la contravarianza “revierte la dirección de la herencia”. Si esta dirección de la herencia la ilustramos con una flecha para que sea más fácil de entender, entonces podemos definir X → Y como que una referencia de la clase X puede ser almacenada en una variable de tipo Y. Podemos concluir entonces que la covarianza “preserva la dirección de la flecha”, por lo que si X → Y, entonces IEnummerable<X> → IEnumerable<Y>, lo que concuerda con lo que hemos visto hasta ahora. Veamos entonces la contravarianza utilizando el delegado Action que esta ligado a este concepto:

Estudiante → Profesor

Action<Estudiante> ← Action<Profesor> (este caso lo vimos en el inciso anterior, en el que utilizamos IComparer que implementa un concepto contravariante)

Action<Action<Estudiante>> → Action<Action<Profesor>>

Action<Action<Action<Estudiante>>> ← Action<Action<Action<Profesor>>>

Funciona no? Utilizar el delegado Action revierte la dirección de la flecha, eso es lo que significa contravarianza. Obviamente, revertir la dirección de la flecha dos veces, es lo mismo que preservar su dirección.

Inciso e: El siguiente código recibe una colección de personas que pueden ejercer cualquier rol, pero se quieren imprimir sólo los que son estudiantes. Complete el espacio para cumplir dicho objetivo.

```
static void PrintStudentsOnly(IEnumerable<object> people)
{
        foreach (var student in people._____________________________________)
                Console.WriteLine(student.Name);
}
```

Respuesta:

```
public static void PrintStudentsOnly(IEnumerable<Persona> people)
{
        foreach (var student in people.OfType<Estudiante>())
                Console.WriteLine(student.Nombre);
}
```

En este caso utilizamos el método OfType, que permite hacer un filtrado en un enumerable dado un tipo específico.