1 Seminario 10 - Dinamismo con C#

1.1 Un corto recorrido por el mundo dinámico de C# 4.0

1.1.1 Round 1 : *dynamic* vs *object*

El tipo ***dynamic*** es un tipo estático presentado por C# 4.0. A pesar de que es un tip estático un objeto de tipo ***dynamic*** omite la comprobación de tipado estático en tiempo de compilación. En la mayoría de los casos, funciona como si tuviera un tipo ***object***. En tiempo de compilación, se supone que un elemento que se escribe como ***dynamic*** admite cualquier operación. Por lo tanto, no tiene que preocuparse por las operaciones o miembros que soporta el objeto marcado como ***dynamic***. Sin embargo, si el código no es válido, se detectan errores en tiempo de ejecución. Algunas ventajas de estos se muestran en el siguiente código.

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            // Asignamos el valor 25 la variable dynamicObject
            // Ahora será objeto de tipo int
            dynamic dynamicObject = 25;
            var sum = dynamicObject + 25;
            Console.WriteLine(sum);

            // Cambiamos el valor la variable dynamicObject
            // Ahora será objeto de tipo string
            dynamicObject = "Hello";
            var message = string.Concat(new string[]{dynamicObject, " ", "World"});
            Console.WriteLine(message);
        }
    }
}

// SALIDA:
// 50
// Hello World
```

Como se puede apreciar el objeto dynamicObject puede comportarse como un objeto de tipo ***int*** en un al principio y luego ser reasignado su otro valor que incluso cambie su tipo. Un comportamiento similar puede ser obtenido usando el tipo ***object***. El código a continuación es equivalente al anterior pero usando el tipo ***object***:

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            object dynamicObject = 25;

            // Casteo a int
            var sum = (int) dynamicObject + 25;
            Console.WriteLine(sum);

            dynamicObject = "Hello";
            // Casteo a string
            var message = string.Concat(new string[]{(string)dynamicObject, " ", "Wor
            Console.WriteLine(message);
        }
    }
}

// SALIDA:
// 50
// Hello World
```

Si bien ambos dan la misma salida eso no quiere decir que se comporten igual. A continuación veremos algunas diferencias:

- Este código de ejemplo no es muy complejo pero vale destacar que cuando usamos ***object*** debemos hacer una casteo cuando queremos usarlo como un tipo específico ya que un objeto tipado con ***object*** no esta libre del chequeo estático de tipos en tiempo de compilación mientras que con ***dynamic*** nos quitamos de arriba este casteo pudiendo tener un código menos verboso pero que puede llegar a ser ilegible si lo escribimos usando malas prácticas. La respuesta a cuál de las dos opciones es mejor solo la situación lo dirá.
- Ambos códigos ejemplos pueden dar error de ejecución si son mal usados, en el caso

de *object* si se hace un casteo erróneo y en el caso de *dynamic* si se le aplica una operación al objeto no acorde al tipo del mismo. Ejemplo:

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            dynamic dynamicObject = new int[10];
            var sum = dynamicObject + 25;
            Console.WriteLine(sum);
        }
    }
}

// SALIDA:
// Unhandled exception. Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: Opera
//    at CallSite.Target(Closure , CallSite , Object , Int32 )
//    at System.Dynamic.UpdateDelegates.UpdateAndExecute2[T0,T1,TRet](CallSite site,
//    at Dynamic.Program.Main(String[] args) in /home/user/Proyectos/RiderProjects/D
```

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            object dynamicObject = new int[10];
            var sum = (int) dynamicObject + 25;
            Console.WriteLine(sum);
        }
    }
}

//Unhandled exception. System.InvalidCastException: Unable to cast *`object`* of type
//   at Dynamic.Program.Main(String[] args) in /home/user/Proyectos/RiderProjects/Dy
```

Como podemos observar los errores son distintos pero el resultado es el mismo, error de

ejecución.

Esto ya nos da una idea de que lo que esta haciendo ***dynamic*** es más que una simple **search and replace** de ***dynamic*** a ***object***, ya que la pila de llamados en los que da error son distintas. Esto tiene sentido ya que el comportamiento a continuación si es muy diferente al anterior.

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            dynamic dynamicObject = new int[10];
            dynamicObject.InventedMethod();
        }
    }
}

// SALIDA
// Unhandled exception. Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'Sys
//    at CallSite.Target(Closure , CallSite , Object )
//    at System.Dynamic.UpdateDelegates.UpdateAndExecuteVoid1[T0](CallSite site, T0
//    at Dynamic.Program.Main(String[] args) in /home/user/Proyectos/RiderProjects/
```

En este caso no podría obtenerse casteando simplemente ***object*** a ***int*** `[]` pues la invocación del método es resuelta en tiempo de compilación y el tipo ***int*** `[]` no posee ningún **InventedMethod**.

Por ultimo mencionar que si se usa ***dynamic*** como parámetro de genericidad como por ejemplo una **`List<dynamic>`** este objeto será declarado en tiempo de ejecución como **`List<object>`**.

```
using System;
using System.Collections.Generic;


namespace Dynamic
{
    internal static class Program
    {
```

```
        private static void Main(string[] args)
        {
            var dynamics = new List<dynamic>();
            Console.WriteLine(dynamics.GetType());
        }
    }
}

// SALIDA
// System.Collections.Generic.List`1[System.Object]
```

1.1.2 Round 2: Python vs C# (Dinamismo)

Perfecto, en este punto ya hemos visto como se comporta un objeto tipado con ***dynamic*** con respecto a ***object***, pero quedan algunas cosa dudas al respecto, ¿será este el límite del dinamismo de `c#`, o tendrá más sorpresas?

Para responder a estas preguntas vamos a fijarnos en `python` el rey de los lenguajes de tipado dinámico. Analicemos el siguiente código escrito en `python3`:

```
class Foo:
    pass

if __name__ == "__main__":
    foo = Foo()
    foo.field1 = "I am a new field"
    print(type(foo))
    print(foo.field1)

# SALIDA
# <class '__main__.Foo'>
# I am a new field
```

Como podemos apreciar la definición de la clase **`Foo`** esta vacía, por lo tanto no posee ni campos ni métodos, creamos una instancia de **`Foo`** llamada `foo` y le asignamos a un nuevo campo `field1` el string `"I am a new fied"` y este es agregado a esta instancia de la clase **`Foo`**, y como podemos comprobar en la salida del programa `foo` sigue siendo una instancia de **`Foo`** pero con un campo no declarado en la clase. Esta característica puede ser bien aprovechada en ocaciones por los programadores de `python`, pero no la podemos lograr con solo declarar una variable como ***dynamic*** en `C#`.

```
using System;
```

```
namespace Dynamic
{
    internal class Foo { }

    internal static class Program
    {
        private static void Main(string[] args)
        {
            dynamic foo = new Foo();
            foo.field1 = "I am a new field";
            Console.WriteLine(foo.field1);
        }
    }
}

// SALIDA
// Unhandled exception. Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'Dyn
//    at CallSite.Target(Closure , CallSite , Object , String )
//    at System.Dynamic.UpdateDelegates.UpdateAndExecute2[T0,T1,TRet](CallSite site
//    at Dynamic.Program.Main(String[] args) in /home/klever/Proyectos/RiderProject
```

Error de ejecución ya que no podemos crearles nuevos campos a un clase ya creada, pero que no cunda el pánico pues ahora veremos como lograr algo similar en C#.

```
using System;
using System.Dynamic;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            dynamic expando = new ExpandoObject();
            expando.field1 = "I am a new field";
            Console.WriteLine(expando.GetType());
            Console.WriteLine(expando.field1);
        }
    }
}
```

```
// SALIDA
// System.Dynamic.ExpandoObject
// I am a new field
```

Para solucionar nuestro problema tenemos a **ExpandoObject**, el cual representa un objeto al que se le pueden agregar y remover campos en tiempo de ejecución de forma dinámica, tambien permite ponerle valores u obtener los mismos. Esta clase soporta binding dinámico, lo que nos permite usar una sintaxis estándar como `foo.sampleMember` en lugar de una sintaxis más compleja como `foo.GetAttribute("sampleMember")`, siempre y cuando usemos el tipo estático ***dynamic***. Además de que podemos iterar por las propiedades de un objeto ya que implementa la interfaz **IDictionary<string, object>** por lo tanto todas la operaciones sobre esta interfaz estan permitidas en un objeto **ExpandoObject**.

```
using System;
using System.Collections.Generic;
using System.Dynamic;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            *`dynamic`* foo = new ExpandoObject();
            foo.field1 = "I am a new field";
            foo.field2 = "I am a second field";
            foreach (var pair in foo)
            {
                Console.WriteLine(pair.Key + " : " + pair.Value);
            }
        }
    }
}
```

```
// SALIDA
// field1 : I am a new field
// field2 : I am a second field
```

Tras ver esto uno puede quedarse satisfecho con las posibilidades que brinda **ExpandoObject** al mundo del dinamismo en C# pero, ¿qué pasa si queremos ir más allá? ¿Qué sucede si queremos controlar el llamado a los miembros de la un tipo dinámico? En este caso **ExpandoObject** no nos será de utilidad ya que es una ***sealed class*** por lo

que no podemos heredar de ella, en cambio tenemos a su hermano **DynamicObject** el cual usaremos para dar solución al siguiente problema.

2 Pregunta I

Implemente el tipo **Prototype** de forma tal que el siguiente código compile y ejecute (C# 4.0) con la salida que se muestra a continuación:

```
using System;

namespace Dynamic
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            dynamic parte1 = new Prototype();
            parte1.MetodoA = (Action<dynamic>) ((self) => { Console.WriteLine("Método

            *`dynamic`* parte2 = new Prototype();
            parte2.MetodoB = (Action<dynamic>) ((self) => { Console.WriteLine("Método

            var obj = parte1.BlendWith(parte2);
            obj.frase = "Hello World!";
            obj.MetodoA();
            obj.MetodoB();
        }
    }
}

// SALIDA:
// MétodoA dice 'Hello World!'
// MétodoB dice 'Hello World!'
```

2.1 Solución a la pregunta 1

2.1.1 "DynamicObject Todo Poderoso hágase en el programa mi voluntad"

Primero declararemos nuestra clase **Prototype** que heredara de **DynamicObject**.

```
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;

namespace Dynamic
{
    public class Prototype: DynamicObject
    {
        private Dictionary<string, dynamic> _memberDictionary = new Dictionary<string

        public override bool TryGetMember(GetMemberBinder binder, out object result)
        {
            return _memberDictionary.TryGetValue(binder.Name, out result);
        }

        public override bool TrySetMember(SetMemberBinder binder, object value)
        {
            _memberDictionary[binder.Name] = value;
            return true;
        }

        public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args
        {
            args = new object[] {this}.Concat(args).ToArray();
            result = _memberDictionary[binder.Name].DynamicInvoke(args);
            return true;
        }

        public Prototype BlendWith(Prototype other)
        {
            var prototype = new Prototype();

            foreach (var pair in _memberDictionary)
                prototype._memberDictionary[pair.Key] = pair.Value;

            foreach (var pair in other._memberDictionary)
                prototype._memberDictionary[pair.Key] = pair.Value;

            return prototype;
```

```
        }
    }
}
```

Pasaremos a explicar cada uno de los métodos sobreescritos en **Prototype** con la siguiente tabla.

Metodo	ón
TryGetMember(GetMemberBinder binder, out object result)	Provee la implementación para la operación de obtener el valor de un miembro. Las clases derivadas de **DynamicObject** pueden sobreescribir este métodos para especificar el comportamiento dinámico para la operación de obtener el valor de una propiedad.
TrySetMember(SetMemberBinder binder, object value)	Provee una implementación para la operación de setear el valor de un miembro. Las clases derivadas de **DynamicObject** pueden sobreescribir este método para especificar elcomportamiento dinámico para la operación de setear el valor de na propiedad.
TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)	Provee una implementación para la operación de invocar a un miembro. Las clases derivadas de **DynamicObject** pueden sobreescribir este método para especificar elcomportamiento dinámico para la operación tales como llamar a un método.

Pueden encontrar las descripciones de más métodos de **DynamicObject** en la documentación online aquí.

Una vez que hemos definido nuestra clase pasaremos a explicar para que usamos cada uno de los métodos y campos que hemos declarado.

```
using System;
// ..

namespace Dynamic
{
    public class Prototype: DynamicObject
    {
        private Dictionary<string, dynamic> _memberDictionary = new Dictionary<string

        // ...
```

```
    }
}
```

En nuestra variable privada `_memberDictionary` guardaremos una el nombre de los miembros que iremos creando en nuestro **`Prototype`** en forma de llave contra valor, donde la llave será el nombre de la propiedad y el valor será el objeto almacenado en dicho miembro.

```
using System;
// ...

namespace Dynamic
{
    public class Prototype: DynamicObject
    {

        // ...

        public override bool TryGetMember(GetMemberBinder binder, out object result)
        {
            return _memberDictionary.TryGetValue(binder.Name, out result);
        }

        // ...

    }
}
```

El método `TryGetMember` se activa cada vez que se produce un dispatch sin asignación en una instancia de nuestro tipo **`Prototype`** siempre que este este declarado con tipo ***`dynamic`***, como es el caso `prototype.frase`, donde en el llamado a nuestro método `TryGetMember` comprobaremos si la llave `"frase"` se encuentra en nuestro diccionario `_memberDictionary` y en caso de este retornaremos `true` y asiganremos a `result` el valor correspondiente a la llave en el diccionario, en caso contrario retornaremos `false` y `result` será `null`. Si el resultado del método es `false` este lanzará el siguiente error **`Microsoft.CSharp.RuntimeBinder.RuntimeBinderException`**.

```
using System;
// ...

namespace Dynamic
```

```
{
    public class Prototype: DynamicObject
    {

        // ...

        public override bool TrySetMember(SetMemberBinder binder, object value)
        {
            _memberDictionary[binder.Name] = value;
            return true;
        }

        // ...

    }
}
```

El método `TrySetMember` se activa cada vez que se produce un dispatch con asignación en una instancia de nuestro tipo **`Prototype`** siempre que este este declarado con tipo ***`dynamic`***, como es el caso `prototype.frase = "Hello World"`, donde el llamado a nuestro método `TrySetMember` siempre retornara `true` ya siempre es posibre asignarle valor a un miembro ya sea existente o nuevo.

```
using System;
// ...

namespace Dynamic
{
    public class Prototype: DynamicObject
    {

        // ...

        public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args
        {
            args = new object[] {this}.Concat(args).ToArray();
            result = _memberDictionary[binder.Name].DynamicInvoke(args);
            return true;
        }

        // ...
```

```
    }
}
```

El método `TryInvokeMember` se activa cada vez que se produce un dispatch con un llamdo a función en una instancia de nuestro tipo **`Prototype`** siempre que este este declarado con tipo ***`dynamic`***, como es el caso `parte1.metodoA()`, donde el llamado a nuestro método `TryInvokeMember` colocará como primer parámetro del vector de argumentos una instancia de si mismo con la palabre clave `this`, ya que todo método dinámico que se declare en **`Prototype`** debe tener como primer parámetro una instancia de él mismo como es el caso de la orden donde tenemos `parte1.MetodoA = (Action<dynamic>) ((self) => { Console.WriteLine("MétodoA dice '{0}'", self.frase); });`, y este puede ser invocado de la siguiente forma `parte1.MetodoA()`, igual a como funciona en `python`.

En este caso una vez colocamos una instancia del objeto como primer elemento del vector de argumentos obtenemos valor del diccionario de `_memberDictionary` asociado al nombre del método, y aqui pueden suceder 4 cosas en tiempo de ejecución:

- si este no existe se lanzará una excepción no controlada del tipo **`System.Collections.Generic.Ke`**
- si este existe pero no es un tipo derivado de la clase **`Delegate`** no podra acceder al método `DynamicInvoke` lanzando la excepción no controlada **`Microsoft.CSharp.RuntimeBinder.RuntimeBinderException`**
- si este existe y es un tipo derivado de la clase **`Delegate`** pero el numero de parámetros no es el correcto se lanzará la excepción no controlada **`System.Reflection.TargetParameterCountException`**
- si este existe, es un tipo derivado de la clase **`Delegate`**, el numero de parámetros es el correcto pero el tipo de los parámetro es incorrecto se lanzará la excepción no controlada **`System.ArgumentException`**

Tras mencionar estos errores puede surgir la duda de porqué querriamos que se lanzara cualquiera de estas excepciones en lugar de capturarlas y retornar `false` en el método. Sucede que si el llamado a `TryInvokeMember` retorna `false` se llama directamente a `TryGetMember` y podrían escaparse casos como `parte1.metodoA(parte1)` donde en el `TryInvokeMember` daría `false`, pero en `TryGetMember` devolveria `true` y si ejecutaria el método, y este comportamiento no es el querido en nuestro tipo **`Prototype`**.

Por último el método `BlendWith` unirá los miembros de ambas partes y creará uno nuevo objeto con la composición de estas.

```
using System;
// ..

namespace Dynamic
```

```
{
    public class Prototype: DynamicObject
    {

        // ...

        public Prototype BlendWith(Prototype other)
        {
            var prototype = new Prototype();

            foreach (var pair in _memberDictionary)
                prototype._memberDictionary[pair.Key] = pair.Value;

            foreach (var pair in other._memberDictionary)
                prototype._memberDictionary[pair.Key] = pair.Value;

            return prototype;
        }

        // ...

    }
}
```

Como se puede observar el código bastante explicito con su funcionamiento. Tan solo creamos un nuevo prototipo y llenamos su diccionario de miembros con las dos partes.

3 Inciso I

¿Por qué los métodos extensores no funcionan con un tipo dinámico *`dynamic`*?

Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original. Los métodos de extensión son una clase especial de método estático, pero se les llama como si fueran métodos de instancia en el tipo extendido. No existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo. Por ejemplo, cualquier tipo que implemente **`IEnumerable<T>`** parecerá tener métodos de instancia como `GroupBy`, `OrderBy`, `Average`, etc.

Los métodos extensores son referenciados en el código IL cuando se compila la instancia

de la clase. O sea al compilar la instancia de la clase, el compilador detecta que esta implementa cierta interfaz o hereda de cierta clase, y le agrega la referencia a los métodos y métodos extensores correspondientes.

Esto no es posible con el tipo *dynamic* pues de la forma en que es asignada es llamando directamente al constructor de una clase, y aunque es posible para el compilador en tiempo de ejecución detectar cuáles y agregar los métodos extensores correspondientes, esto sería muy costoso pues tendría que añadir recursivamente todos los namespaces que utilice la clase. Dicho de otra forma, se tendría que agregar el namespace donde esta implementada y a su vez todos los que esta utilice, todos los que estas utilicen y así sucesivamente.

Eric Lippert, Desarrollador y MVP de C# responde a la siguiente pregunta en StackOverflow el 15 de marzo de 2011.

¿Por qué los métodos extensores no funcionan con un tipo dinámico *dynamic*?

"...los métodos extensores en código no dinámico funcionan haciendo una búsqueda completa en todas las clases conocidas por el compilador hasta encontrar una clase estática que tenga el método extensor que coincida. La búsqueda va en orden basada en la jerarquía de los namespaces y las directivas using en cada namespace. Esto significa que para que la invocación de un método de extensión dinámico sea resuelto correctamente, el DLR tiene que conocer en tiempo de ejecución (runtime) cuales es la jerarquía de namespaces y directivas using estaban en el código fuente. Nosotros no tenemos un mecanismo manuable para codificar toda la información necesaria en el lugar de la llamada (call site). Nosotros consideramos inventar ese mecanismo, pero decidimos que era de muy alto costo y producía mucho riesgo en la planificación para que valiera la pena."

4 Inciso II

Implemente también para `Prototype` un método `Clone` de manera que los miembros de tipo función que fuesen adquiridos por el original o alguna de sus copias sean compartidos entre todos.

4.1 Solución al Inciso II

4.1.1 "Y el programador dijo 'Hagase el clon!!!' y el clon se hizo sin más explicación, pero ... ¿qué tipo de clon era?"

Para dar solución a este inciso primero hablaremos del patrón de diseño Prototype, de los tipos de copias que podemos implementar y cuáles son las ventajas y desventajas de las

mismas.

4.1.1.1 Patrón de Diseño Prototype

Prototype es un patrón de diseño creacional que permite clonar objetos sin acoplarlos a su especificas. Todas las clases de prototipos deben tener una interfaz comun que permita copiar objetos incluso si se desconocen sus clase concretas. Los objetos prototipos pueden crear copias completas, ya que los objetos de la misma clase pueden acceder a los campos privados de cada uno.

4.1.1.2 Tipos de copias

Analizaremos las copias superficiales de objetos a las que nos referiremos como shallow copies y las copias profundas a las que nos referiremos como deep copies en adelante.

Tipo de copia	Descrición
Shallow Copy	La shallow copy crea un clon del objeto cuyos campos tienen el mismo valor que los del objeto original, y en el caso de alguno de eso campos sea un tipo por referencia se copiara solo la referencia, por lo que tanto el campo del objeto original como el del clon estarán apuntando al mismo ojeto.
Deep Copy	La deep copy crea un clon del objeto cuyos campos tienen el mismo valor que los del objeto original, pero en el caso de alguno de eso campos sea un tipo por referencia se creará una copia de este campo con los mismos valores, por lo que tanto el campo del objeto original como el del clon tendrían los mismos valores pero serían objetos distintos.

Como podemos apreciar una deep copy es más “copia” que una shallow copy, en este caso solo analizaremos las deep copies de un solo nivel ya que es posible hacer deep copy recursivamente pero esto podria traer problemas en alguna situaciones, como por ejemplo

cuando un objeto se autoreferencia. El siguiente caso en un ejemplo sencillo pero claro:

```
using System;

namespace Dynamic
{
    internal class Foo : ICloneable
    {
        public Foo auto;

        public object Clone()
        {
            return new Foo {auto = (Foo) auto.Clone()};
        }
    }

    internal static class Program
    {

        private static void Main(string[] args)
        {
            var foo = new Foo();
            var foo2 = new Foo();

            foo.auto = foo2;
            foo2.auto = foo;

            var foo3 = foo.Clone();
        }
    }
}

// SALIDA
// Stack overflow.
```

Moraleja : NO SER TAN EXAGERADO CON LAS CLONACIONES

Veamos ahora como ponemos en práctica el método `Clone` de **`Prototype`**. Primero crearemos una interfaz **`ICopiable`** que tendrá la definición de los métodos `DeepCopy` y `ShallowCopy`

```
namespace Dynamic
{
```

```
    public interface ICopiable
    {
        object ShallowCopy();
        object DeepCopy();
    }
}
```

Ahora haremos que **Prototype** implemente **ICloneable** e **ICopiable**

```
using System;
// ...

namespace Dynamic
{
    public class Prototype: DynamicObject, ICloneable, ICopiable
    {

        // ...

        public object Clone()
        {
            return ShallowCopy();
        }

        public object ShallowCopy()
        {
            return MemberwiseClone();
        }

        public object DeepCopy()
        {
            var prototype = (Prototype) MemberwiseClone();
            prototype._memberDictionary = new Dictionary<string, dynamic>(_memberDict
            return prototype;
        }
    }
}
```

C# ya viene preparado con el método `MemberwiseClone` que crea una shallow copy del objeto que lo invoca. Este crea un nuevo objeto y luego copia los campos no estáticos del objeto original. Si el campo es por valor de hace uan copia del mismo bit a bit, de lo contrario, si es por referencia, solo se copia la referencia de tal forma que ambos campos

apuntan al mismo opjeto.

Por otro lado nuestra implementación de `DeepCopy` crea tambien una copia del diccionario de miembros con los mismos valores que el original pero ahora ambos campos no apuntan al mismo objeto. Sin embargo los tipos por referencia del nuevo diccionario y el anterior si son iguales ya que hicimos una deep copy de un solo nivel.

4.1.1.3 ¿Cuál usar en `Clone`?

En `Clone` usaremos `ShallowCopy` ya que, aunque ambas siguen el comportamiento que pide la orden, hacer una shallow copy en menos costoso que hacer una deep copy y consume menos recursos. Reflejando aqui las ventajas y desventajas de cada uno, mientras que deep copy crea una copia más independiente del objeto original esta puede ser muy costosa en tiempo y en recursos, En cambio una shallow copy no independizará completamente al clon del original pero es menos costosa en tiempo y recursos.

5 Referencias

- Microsoft. (20 de 07 de 2015). docs.microsoft.com. Obtenido de docs.microsoft.com: https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/extension-methods
- Valle, M. d. (25 de 03 de 2011). Tipado dinámico en C# 4.0. Obtenido de Tipado dinámico en C# 4.0: https://desarrolloweb.com/articulos/tipado-dinámico-c-dotnet.ht

6 Fin