

PROGRAMACIÓN ORIENTADA A OBJETOS

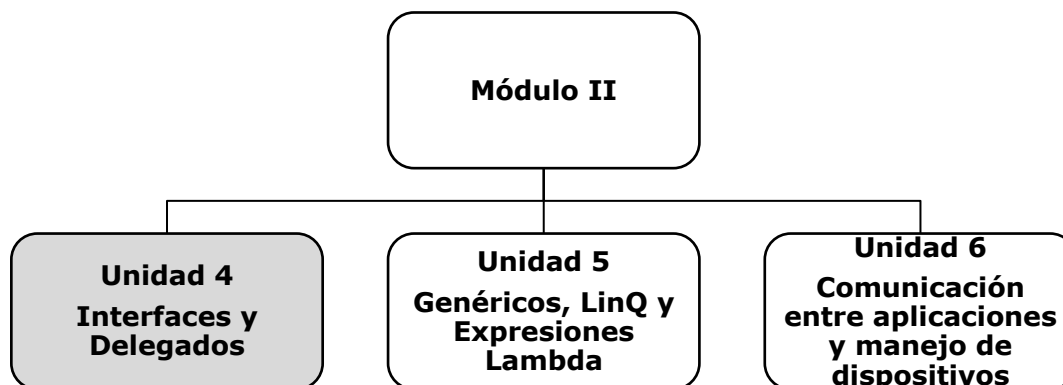
Módulo II

Programación de aplicaciones utilizando la técnica de la Programación Orientada a Objetos

Unidad 4

Interfaces y Delegados

Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci



Presentación

En esta unidad veremos todo lo necesario para poder trabajar con Interfaces y Delegados.

Esto nos permitirá construir programas más robustos y flexibles.

Por todo lo expresado a través del estudio de esta unidad, usted logrará:

- Comprender las ventajas de utilizar **interfaces**.
- Identificar, distinguir e implementar las interfaces **Comparable**, **Comparable**, **Cloneable**, **Comparable** e **Comparable**.
- Utilizar delegados en la programación para lograr asignaciones dinámicas.

Los siguientes **contenidos** conforman el marco teórico y práctico que contribuirá a alcanzar las metas de aprendizaje propuestas:

Interfaces. Desarrollo e Implementación de una interfaz.

La interfaz Comparable. La interfaz Comparable. La interfaz Cloneable. Las interfaces Comparable e Comparable.

Delegados. Delegados usando métodos con nombre. Delegados con métodos anónimos.

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Contenidos y Actividades

1. INTERFACES



Lectura requerida

- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/>
- [https://msdn.microsoft.com/es-es/library/ms173156\(v=vs.80\).aspx](https://msdn.microsoft.com/es-es/library/ms173156(v=vs.80).aspx)
- [https://msdn.microsoft.com/es-es/library/system.collections.icomparer\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.collections.icomparer(v=vs.110).aspx)
- [https://msdn.microsoft.com/es-es/library/system.icomparable\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.icomparable(v=vs.110).aspx)
- <https://msdn.microsoft.com/es-es/library/system.icloneable%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396>
- [https://msdn.microsoft.com/es-es/library/system.collections.ienumerable\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.collections.ienumerable(v=vs.110).aspx)
- [https://msdn.microsoft.com/es-es/library/system.collections.ienumerator\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.collections.ienumerator(v=vs.110).aspx)

2. DELEGADOS



Lectura requerida

- <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/delegates/>
- <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/delegates/using-delegates>

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. INTERFACES

Las **interfaces** se utilizan para poder definir propiedades, métodos y eventos que luego deberán implementar aquellas clases que implementen la interfaz.

Esto hace que las clases que implementan una misma **interfaz**, deban implementar todo aquello que esté definido en la **interfaz** implementada. Esto implica que tendrán las propiedades, métodos y eventos que la interfaz tenga definida y por lo tanto coincidirán en las firmas de estos miembros.

Considerando lo expuesto, podemos decir que las interfaces ayudan a homogeneizar. Otra ventaja de utilizar interfaces es que las mismas **tipan**. Esto significa que cuando una clase implementa una interfaz también adquiere un nuevo **tipo**, que es el **tipo** de la interfaz. De esta manera, se obtiene una forma muy flexible de compartir un **tipo común** entre las clases que se encuentran en distintas jerarquías de herencia. Si recordamos el concepto de polimorfismo abordado con anterioridad, el mismo se sustentaba en la posibilidad de disponer de un **tipo** común y hasta el momento solo lo sabíamos construir por medio de una jerarquía de herencia. Ahora poseemos otro mecanismo para obtener ese tipo común y es haciendo que todas las clases que lo necesiten implementen la misma interfaz, sin necesariamente pertenecer a la misma jerarquía de herencia.

Como se puede observar este mecanismo es una alternativa altamente flexible a la **herencia**.

Por otro lado y como ya hemos mencionado, las interfaces tipan a las clases que las implementan. Considerando que potencialmente podríamos consultar si un objeto es de un tipo determinado, entonces podremos consultar si un objeto es del tipo de una determinada interfaz. Si la respuesta es afirmativa, la única manera que ese objeto sea del tipo de la interfaz, es que la clase desde donde él fue instanciado tenga implementada esa interfaz. Siguiendo esta línea de razonamiento, si esa clase tiene implementada esa interfaz entonces tiene la obligación de tener implementado todo lo que esa interfaz define, caso contrario el framework daría un error. Esto nos conduce a tener certeza sobre que miembros (propiedades, eventos y métodos) posee ese objeto en particular y en función a ello, se puede enviar un mensaje con la certeza que tendrá una resolución.

Este último concepto permite abordar la idea de, "programar hoy una clase para interactuar en un futuro con clases que hoy aún no existen". ¿Cómo se puede realizar esto? Sencillo, si desarrollamos una clase "A" que le envíe un mensaje "M" a otra clase "B". Luego exigimos que la clase "B", implemente la interfaz "I". Si la interfaz "I" posee definido el método "M", no le queda otra posibilidad a la clase "B" que implementarlo. De esta manera la clase "A" puede enviarle el mensaje "M" a "B", con la seguridad que "B" recepcionará y responderá adecuadamente al mensaje "M".

El framework .Net aprovecha mucho esta última característica. Es por ello, que hay desarrolladas un número muy importante de interfaces. Además prevé la posibilidad que podamos desarrollar nuestras propias interfaces personalizadas.

Para comprender más profundamente el tema, analizaremos cinco interfaces del framework y luego construiremos una propia.

Las cinco interfaces del framework que veremos son:

IComparable

IComparer

ICloneable

IEnumerable

IEnumerator

ICOMPARABLE

La interfaz **IComparable** se utiliza, entre otras cosas, para poder interactuar con el método **"Sort"** de la clase **"Array"** y lograr que un **"Array"** de un determinado tipo pueda ser ordenado por un criterio peculiar.

La forma de implementar la interfaz en una clase es colocar dos puntos ":" luego del nombre de la clase y el nombre de la interface.

```
public class Persona : IComparable
{
    2 referencias
    public string Nombre { get; set; }
    0 referencias
    public string Apellido { get; set; }
    0 referencias
    public int Edad { get; set; }

    0 referencias
    public int CompareTo(object obj)
    {
        return String.Compare(((Persona)obj).Nombre, this.Nombre);
    }
}
```

EJ0001

Esto generará dentro de la clase la necesidad de implementar lo que ella define, en este caso es el método **CompareTo**.

El valor de retorno de **CompareTo** es un entero. El significado del valor retornado es el siguiente:

Valor	Significado
Menor que cero	La instancia actual precede al objeto especificado por el parámetro obj de método CompareTo .
Cero	La instancia actual ocupa la misma posición en el criterio de ordenación que el objeto especificado por el parámetro obj del método CompareTo .
Mayor que cero	La instancia actual se ubica en una posición posterior al objeto especificado por el parámetro obj del método CompareTo .

La idea es colocar dentro de la función **CompareTo** el código que permite determinar el valor retornado y en consecuencia el ordenamiento en cuestión.

Siempre se estarán comparando dos objetos, uno es el objeto que se está ejecutando y el otro es el que se envía por parámetro a la función **CompareTo**.

El ejemplo **EJ0001** muestra la manera de ordenar un array utilizando el método **Sort**. En este ejemplo el array de personas es ordenado por nombre. Para que esto sea posible en la clase **Persona** se implementó la interface **IComparable**. En método **CompareTo** compara el objeto que actualmente se está ejecutando (**this**), con el objeto recibido en el parámetro **obj**. Para lograrlo se aprovechó el método **Compare** de la clase **string**. Este método compara dos objetos **string** especificados y devuelve un entero que indica su posición relativa en el criterio de ordenación. La tabla de salida de este método es coincidente con lo requerido por el **CompareTo**. Es por ello que se retorna directamente el valor que entrega. La tabla de **String.Compare** es:

Valor	Condición
Menor que cero	La primera subcadena precede a la segunda subcadena en el criterio de ordenación.
Cero	Las subcadenas aparecen en la misma posición en el criterio de ordenación, o <i>length</i> es cero.
Mayor que cero	La primera subcadena sigue a la segunda subcadena en el criterio de ordenación.

La implementación del ejemplo se realiza creando un array **P** y luego ordenándolo como se muestra a continuación.

```

private void Form1_Load(object sender, EventArgs e)
{
    textBox1.Clear();
    Persona[] P = { new Persona() { Nombre="Juan",Apellido="Garcia",Edad=22},
                    new Persona() { Nombre="Ana",Apellido="Kardo",Edad=34},
                    new Persona() { Nombre="Cecilia",Apellido="Perez",Edad=29},
                    new Persona() { Nombre="Leonardo",Apellido="Romano",Edad=41}};

    textBox1.Text = "Antes de Ordenar: " + Environment.NewLine + Environment.NewLine;
    foreach (Persona xP in P)
    { textBox1.Text += "\t" + xP.Nombre + "\t" + xP.Apellido + "\t" + xP.Edad.ToString() + Environment.NewLine; }
    Array.Sort(P);
    textBox1.Text += Environment.NewLine + Environment.NewLine + "Despues de Ordenar: " +
                    Environment.NewLine + Environment.NewLine;
    foreach (Persona xP in P)
    { textBox1.Text += "\t" + xP.Nombre + "\t" + xP.Apellido + "\t" + xP.Edad.ToString() + Environment.NewLine; }
    textBox1.SelectionStart = textBox1.Text.Length;
}

```

EJ0001

En el ejemplo se puede observar que el parámetro **obj** es de tipo **object**, lo que fuerza a que se tenga que cambiar al tipo **Persona**.

```

return String.Compare(this.Nombre, ((Persona)obj).Nombre);

```

Para evitar esto se puede utilizar la versión genérica de **IComparable**. Esto haría que la clase **Persona** quede como se muestra en el ejemplo **EJ0002**.

```

public class Persona : IComparable<Persona>
{
    8 referencias
    public string Nombre { get; set; }
    6 referencias
    public string Apellido { get; set; }
    6 referencias
    public int Edad { get; set; }

    0 referencias
    public int CompareTo(Persona other)
    {
        return String.Compare(this.Nombre, other.Nombre);
    }
}

```

EJ0002

Una característica de esta interfaz es que permite ordenar por un solo criterio. Si se desea ordenar en forma descendente, se deberá invertir el resultado como muestra el ejemplo **EJ0003**.

```
public class Persona : IComparable<Persona>
{
    8 referencias
    public string Nombre { get; set; }
    6 referencias
    public string Apellido { get; set; }
    6 referencias
    public int Edad { get; set; }

    0 referencias
    public int CompareTo(Persona other)
    {
        return (String.Compare(this.Nombre, other.Nombre)) * -1;
    }
}
```

EJ0003

Y si deseamos ordenar por **Apellido** se perdería la posibilidad de ordenar por **Nombre**, como muestra el ejemplo **EJ0004**.

```
public class Persona : IComparable<Persona>
{
    6 referencias
    public string Nombre { get; set; }
    8 referencias
    public string Apellido { get; set; }
    6 referencias
    public int Edad { get; set; }

    0 referencias
    public int CompareTo(Persona other)
    {
        return String.Compare(this.Apellido, other.Apellido);
    }
}
```

EJ0004

ICOMPARER

Como hemos podido observar, en la interfaz anterior solo podemos ordenar por un criterio simultáneamente. Pero hay ocasiones en las que deseamos ordenar por más de un criterio a la vez. En ese caso utilizamos la interfaz **Icomparer**.

Esta interfaz puede con clases anidadas. Cada clase anidada tendrá implementada la interfaz **Icomparer** y en cada una de ellas se definirá un criterio particular de ordenamiento, pudiendo tener tantos como clases anidadas coloquemos.

Esta interfaz define el método **Compare**, el cual posee dos parámetros **X** e **Y**. Cada parámetro representa a un objeto y la comparación se realiza entre ellos.

Los valores retornados por el método **Compare** son:

- a. Si el valor es menor que cero significa que **X** es menor que **Y**.
- b. Si el valor es cero significa que **X** es igual a **Y**.
- c. Si el valor es mayor que cero significa que **X** es mayor que **Y**.

En los siguientes fragmentos de código correspondientes al ejemplo **EJ0005**, se puede observar como la clase **Persona** posee seis clases anidadas:

NombreASC

NombreDESC

ApellidoASC

ApellidoDESC

EdadASC

EdadDESC

Cada una de estas implementa la interfaz **Icomparer** y en el método **Compare** implementa el criterio de orden correspondiente.

```

public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    public int Edad { get; set; }
    public class NombreAsc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y){ return String.Compare(x.Nombre, y.Nombre); }}
    public class NombreDesc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y){ return String.Compare(x.Nombre, y.Nombre) * -1; }}
    public class ApellidoAsc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y){ return String.Compare(x.Apellido, y.Apellido); }}
    public class ApellidoDesc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y) { return String.Compare(x.Apellido, y.Apellido) * -1; }}
    public class EdadAsc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y)
        {
            int rdo = 0;
            if (x.Edad < y.Edad) rdo = -1;
            if (x.Edad == y.Edad) rdo = 0;
            if (x.Edad > y.Edad) rdo = 1;
            return rdo;
        }
    }
    public class EdadDesc : IComparer<Persona>
    {
        public int Compare(Persona x, Persona y)
        {
            int rdo = 0;
            if (x.Edad < y.Edad) rdo = -1;
            if (x.Edad == y.Edad) rdo = 0;
            if (x.Edad > y.Edad) rdo = 1;
            return rdo * -1;
        }
    }
}

```

EJ0005

A continuación, se puede observar como utilizar una de las sobrecargas del método **Sort** para ordenar un array de personas.

El método **Sort** solicita como primer parámetro un array, que es el array a ordenar. Como segundo parámetro un objeto del tipo **IComparer**, que es el que posee la implementación del método **Comparer** y el criterio de ordenamiento. A partir del criterio de ordenamiento se decide retornar un entero menor que cero, igual a cero o mayor que cero.

La siguiente línea de código del ejemplo **EJ0005** ejemplifica lo expresado:

```
Array.Sort(P, new Persona.NombreAsc());
```

P: Representa al array que posee a las personas.

new Persona.NombreAsc(): Representa una instancia de la clase **NombreAsc** que está anidada en la clase **Persona** y posee implementada la interface **IComparer**.

IENUMERABLE e IENUMERATOR

La interfaz **IEnumerable** expone un enumerador, que admite una iteración simple sobre una colección. Esta interfaz propone que se implemente el método **GetEnumerator** el que retorna un objeto de tipo **IEnumerator**.

```
IEnumerator IEnumerable.GetEnumerator()
```

La interfaz **IEnumerator** obliga a implementar en la clase que la implementa, la siguiente propiedad y los dos métodos que a continuación se exponen:

Current	Propiedad de solo lectura. Retorna el elemento que se está recorriendo.
Reset	Método. Se inicializan valores que se utilizan en la iteración.
Move	Método. Determina que elemento se va a colocar en Current y retorna verdadero. Si ya no hay nada por recorrer debe retornar falso.

Para comprender su funcionamiento supondremos que existe una especificación técnica que requiere lo siguiente:

Se posee una lista de productos que al iterarlos con un foreach, cada uno retorna los valores correspondientes a su código EAN-13.

Antes de avanzar se explicará qué es un código EAN-13. El código EAN-13 está constituido por trece (13) dígitos y con una estructura dividida en cuatro partes. De izquierda a derecha se debe interpretar como:

Código del país: en donde radica la empresa, compuesto por tres (3) dígitos.

Código de empresa: es un número compuesto por cuatro dígitos, que identifica al propietario de la marca. Es asignado por la asociación de fabricantes y distribuidores.

Código de producto: siguiente cinco dígitos.

Dígito de control: un dígito. Para comprobar los 12 dígitos anteriores.

Para calcularlo, numeramos los dígitos de derecha a izquierda comenzando con 1. A continuación se suman los dígitos de las posiciones impares, el resultado se multiplica por 3, y se le suman los dígitos de las posiciones pares. Se busca decena inmediatamente superior y se le resta el resultado obtenido. El resultado final es el dígito de control. Si el resultado es múltiplo de 10 el dígito de control será cero (0).

Por ejemplo:

EAN-13 7791234567805

Código de país: 779

Código de empresa: 1234

Código de producto: 56780

Dígito de control: 5

7	7	9	1	2	3	4	5	6	7	8	0		
12	11	10	9	8	7	6	5	4	3	2	1		
	7	+	1	+	3	+	5	+	7	+	0	=	23
Se multiplica $23 * 3 = 69$													
7	+	9	+	2	+	4	+	6	+	8		=	36
Se suma $69 + 36 = 105$													
Se busca el múltiplo de diez inmediatamente superior a 105 que es 110													
Se realiza la resta $110 - 105 = 5$													
En resultado 5 es el dígito de control													

En el ejemplo **EJ0006** se observa que la clase **Producto** implementa las interfaces **IEnumerable** e **IEnumerator**. Esto provoca que cada objeto **Producto** sea del tipo **IEnumerable** e **IEnumerator**. Cuando a un producto se lo itera con un **foreach**, este indaga al objeto sobre si su tipo es **IEnumerable**. En caso afirmativo, le envía un mensaje **GetEnumerator()** al objeto **Producto** y este le retorna una referencia de sí mismo (**this**).

Sobre esta referencia el **foreach** envía un mensaje **Move**. Si la respuesta a este mensaje es **true**, significa que se colocó algo que se puede obtener mediante la propiedad **Current**. En este caso el **foreach** envía el mensaje **Current** y obtiene lo que se encuentra almacenado allí. El **foreach** lo consume y en este ejemplo se muestra en la caja de texto. Luego vuelve a enviar un mensaje **Move** y así sucesivamente hasta que se obtenga un **false**. En este caso el **foreach** deja de consultar la propiedad **Current** y finaliza su ejecución.

A continuación se expone la clase **Producto**.

```
public class Producto : IEnumerator, IEnumerable
{
    string _Current = "";
    public int Codigo { get; set; }
    public string Descripcion { get; set; }
    public string EAN13 { get; set; }
    public object Current => _Current;
    static int c = 0;
    bool r = false;
    public bool MoveNext()
    {
        if (c == 0) { _Current = "País: " + EAN13.Substring(0, 3); r = true; c++; }
        else if (c == 1) { _Current = "Empresa: " + EAN13.Substring(3, 4); r = true; c++; }
        else if (c == 2) { _Current = "Producto: " + EAN13.Substring(6, 5); r = true; c++; }
        else if (c == 3) { _Current = "Dígito Verificador: " + EAN13.Substring(12, 1); r = true; c++; }
        else { Reset(); }
        return r;
    }
    public void Reset() { c = 0; r = false; _Current = ""; }
    public IEnumerator GetEnumerator()
    {
        return this;
    }
}
```


EJ0006

En el siguiente código del mismo ejemplo se puede ver como se recorre una lista de productos y se itera a cada **Producto** para obtener los datos del código EAN-13.

```
List<Producto> _ListaProducto = new List<Producto>();
private void button1_Click(object sender, EventArgs e)
{
    _ListaProducto.AddRange( new Producto[] { new Producto() { Codigo = 1, Descripcion = "Tuerca", EAN13 = "7791234567895" },
                                              new Producto() { Codigo = 2, Descripcion = "Arandela", EAN13 = "7793654098716" },
                                              new Producto() { Codigo = 3, Descripcion = "Tornillo", EAN13 = "7798258013769" } });
    textBox1.Clear();
    foreach (Producto P in _ListaProducto)
    {
        foreach (string S in P) { textBox1.Text += S + Environment.NewLine; }
        textBox1.Text+=Environment.NewLine;
    }
}
```

EJ0006

Lo que se obtiene es:



The screenshot shows a web interface with a button labeled 'Ver' on the left. To the right of the button, there are three blocks of text, each representing a set of data. The first block contains: País: 779, Empresa: 1234, Producto: 45678, and Dígito Verificador: 5. The second block contains: País: 779, Empresa: 3654, Producto: 40987, and Dígito Verificador: 6. The third block contains: País: 779, Empresa: 8258, Producto: 80137, and Dígito Verificador: 9.

País	Empresa	Producto	Dígito Verificador
779	1234	45678	5
779	3654	40987	6
779	8258	80137	9

EJ0006

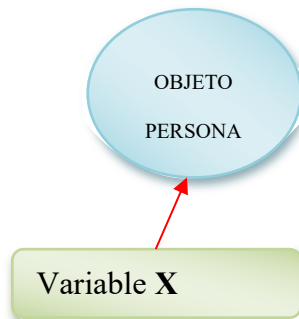
ICLONEABLE

La interfaz **ICloneable** permite realizar la clonación de objetos. Crea una nueva instancia de una clase con el mismo estado de una instancia existente. Esta interfaz contiene un miembro, el método **Clone**, que está diseñado para proporcionar compatibilidad más allá de la proporcionado por el método **MemberwiseClone** que provee **Object**. La clonación se puede realizar de dos maneras diferentes. Una es la **clonación superficial** y la otra la **clonación profunda**.

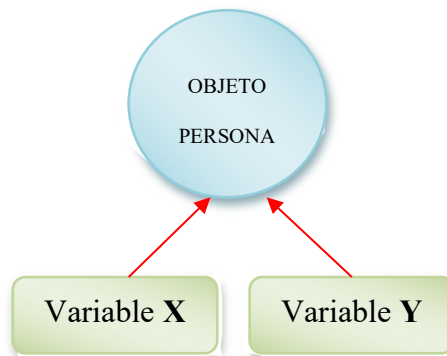
Clonación Superficial

El método **MemberwiseClone** crea una copia superficial, crea un nuevo objeto ,y a continuación, copiar los valores de los campos no estáticos del objeto actual al nuevo objeto. Si un campo es un tipo de valor, se realiza una copia bit a bit del campo. Si un campo es un tipo de referencia, la referencia se copia, pero el objeto al que se hace referencia no, por lo tanto, el objeto original y su clon hacen referencia al mismo objeto.

La siguiente representación permite diferenciar una asignación de una clonación superficial. Supongamos que una variable X apunta a un objeto:

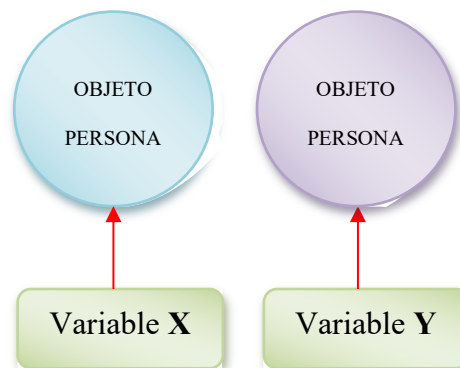


Luego asignamos la variable X a otra variable Y, por ejemplo $Y=X$. La representación sería la siguiente:



En este escenario las dos variables apuntan al mismo objeto, esto significa que cualquier cambio que se realice a través de la variable **X** afecta al único objeto que tenemos en memoria. Esto causaría que el cambio se vea reflejado si consultamos u operamos al objeto desde la variable **Y**.

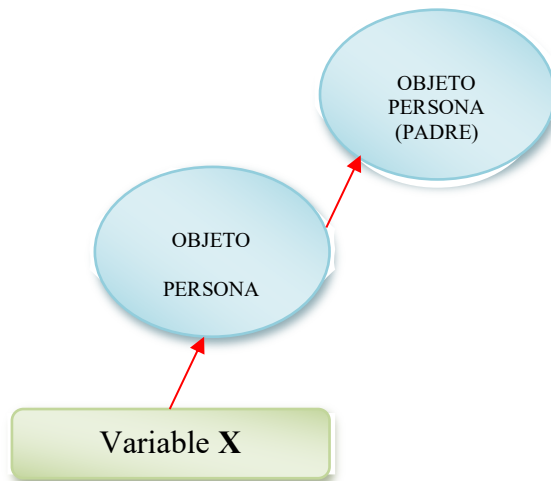
En caso de realizar una clonación superficial del tipo **Y=X.clone()**, la representación se vería como se muestra a continuación:



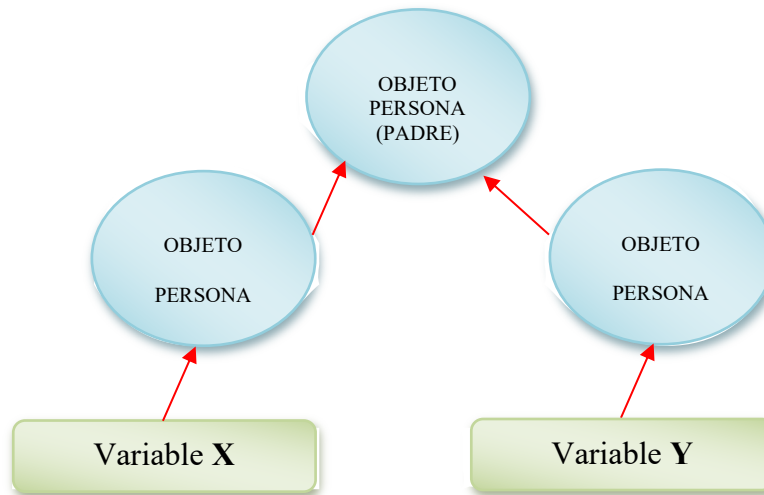
Claramente se puede observar como en este último caso se genera un objeto adicional en memoria. Este objeto apuntado por la variable **Y** es independiente del apuntado por la variable **X**. Al momento de constituirlo, si bien los objetos son diferentes e independientes poseen el mismo **estado**. Los cambios que se realicen al objeto apuntado por **X**, a través de **X**, no afectarán al objeto apuntado por **Y** y viceversa.

No obstante, si bien las posibilidades de la clonación superficial brindan una funcionalidad interesante, la misma posee características que en determinados escenarios se constituyen o visualizan como limitaciones. Se ha mencionado al principio de la exposición sobre la clonación superficial que: "Si un campo es un tipo de referencia, la referencia se copia, pero el objeto al que se hace referencia no, por lo tanto, el objeto original y su clon hacen referencia al mismo objeto".

Supongamos que tomando el ejemplo anterior una persona a su vez apunta a otra persona. Esto podría darse, por ejemplo, porque el objeto define una relación de asociación "tiene un padre". También asumamos que el campo que mantiene la referencia al objeto que representa al padre se denomina **Padre**. Lo expuesto podría visualizarse esquemáticamente de la siguiente manera:



Al realizar la clonación superficial **Y=X.clone()** se vería de la siguiente forma:



Al clonar X se produce el mismo efecto antes expresado, pero como el campo **Padre** mantiene una referencia a otro objeto persona, se copia la referencia y no se clona al padre. Como lo mencionamos, esto en algunos casos puede significar una limitante. Más adelante lo solucionaremos con un técnica de clonación denominada **clonación profunda**.

Por ahora nos concentraremos en el mecanismo que opera cuando realizamos una clonación superficial. Para comprenderlo en profundidad analizaremos el ejemplo **EJ0007**. En el mismo existe una clase **Empleado**. Esta clase posee una referencia a otro **Empleado** que representa a su jefe. La referencia mencionada se administra a través de la propiedad autoimplementada denominada **Jefe**. También se puede observar la implementación de la interface **ICloneable**, la que obliga a que se implemente el método **Clone()**. Es método aplica el mencionado **MemberwiseClone**, para lograr la clonación superficial.

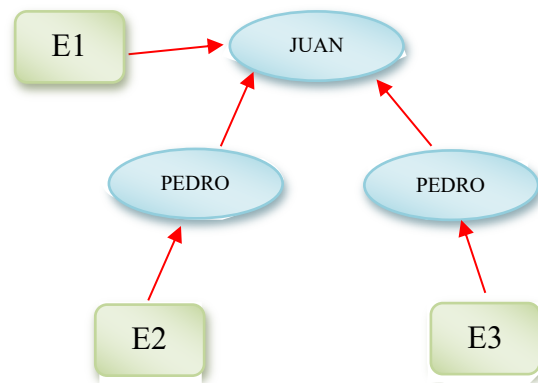
```

public class Empleado : ICloneable
{
    15 referencias
    public string Nombre { get; set; }
    8 referencias
    public Empleado Jefe { get; set; }

    1 referencia
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
  
```

EJ0007

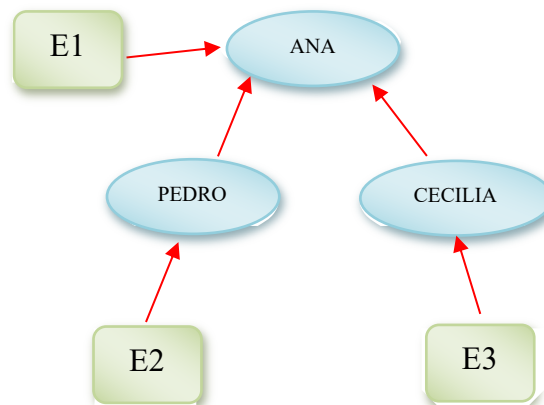
El siguiente código crea un empleado **E1** llamado **Juan** que no posee Jefe. Luego crea otro empleado **E2** llamado **Pedro**, que define a **E1**(Juan) como su jefe. Luego declara una variable **E3** y le asigna un clon de **E2**. Esquemáticamente se vería como:



```
Empleado E1 = new Empleado() { Nombre = "Juan", Jefe = null };  
Empleado E2 = new Empleado() { Nombre = "Pedro", Jefe = E1 };  
Empleado E3 = (Empleado)E2.Clone();
```

EJ0007

Luego el ejemplo muestra estos valores y propone cambiarle el nombre al jefe de **E3** y a **E3**. Supongamos que al jefe de **E3** se le coloca Ana y a **E3** Cecilia. El esquema quedaría como:



El ejemplo avanza exponiendo los nombres y demuestra que los nombres de los objetos apuntados por **E2** y **E3** funcionan de manera independiente. No ocurre lo mismo al modificar el nombre de jefe de **E3**, ya que también se modifica el nombre del jefe de **E2**, debido a la forma de trabajar de la clonación superficial. A continuación se observa el código del ejemplo **EJ0007** que realiza lo antes mencionado y más adelante se procede a **explicar la clonación** profunda que soluciona este efecto no deseado.

```
private void button1_Click(object sender, EventArgs e)
{
    Empleado E1 = new Empleado() { Nombre = "Juan", Jefe = null };
    Empleado E2 = new Empleado() { Nombre = "Pedro", Jefe = E1 };
    Empleado E3 = (Empleado)E2.Clone();

    textBox1.Clear();
    textBox1.Text += "E1 es jefe de E2" + Environment.NewLine + Environment.NewLine;
    textBox1.Text += "E1 Nombre: " + E1.Nombre + Environment.NewLine;
    textBox1.Text += "E2 Nombre: " + E2.Nombre + " - El nombre del Jefe de E2: " + E2.Jefe.Nombre +
        Environment.NewLine + Environment.NewLine;
    textBox1.Text += "E3 es un clon de E2" + Environment.NewLine;
    textBox1.Text += "E3 Nombre: " + E3.Nombre + " - El nombre del Jefe de E3: " + E3.Jefe.Nombre +
        Environment.NewLine + Environment.NewLine;
    textBox1.Text += "Se procede a cambiarle el nombre al jefe de E3" + Environment.NewLine;
    E3.Jefe.Nombre = Interaction.InputBox("Ingrese el nuevo nombre para el jefe de E3: ") + Environment.NewLine;
    textBox1.Text += "E3 Nombre: " + E3.Nombre + " - El nombre del Jefe de E3: " + E3.Jefe.Nombre +
        Environment.NewLine + Environment.NewLine;
    textBox1.Text += "Se procede a cambiarle el nombre a E3" + Environment.NewLine;
    E3.Nombre = Interaction.InputBox("Ingrese el nuevo nombre para E3: ");
    textBox1.Text += "E3 Nombre: " + E3.Nombre + " - El nombre del Jefe de E3: " + E3.Jefe.Nombre +
        Environment.NewLine + Environment.NewLine;
    textBox1.Text += "Observar a continuación lo que ocurrió. El nombre de E2 se ha conservado a pesar de " +
        "haber cambiado el nombre de E3 Sin embargo el cambio del nombre del jefe de E3 afectó el " +
        "nombre del jefe de E2." + Environment.NewLine + Environment.NewLine;
    textBox1.Text += "E2 Nombre: " + E2.Nombre + " - El nombre del Jefe de E2: " + E2.Jefe.Nombre +
        Environment.NewLine + Environment.NewLine;
    textBox1.Text += "Esto es debido a que la clonación realizada es superficial. " +
        "Se puede solucionar esto con una clonación profunda." + Environment.NewLine;
}
```

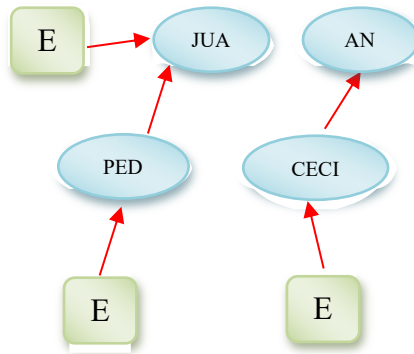
EJ0007

Clonación profunda

La clonación profunda se logra aplicando técnica de programación. La misma es el resultado de realizar una clonación superficial y sobre esa copia, buscar todas las referencias, para en cada una de ellas clonarlas recursivamente.

Con esta técnica se logrará clonar todo el grafo de objetos. En nuestro ejemplo ese grafo estaría compuesto por el empleado y su jefe. Pero en caso que el jefe posea un jefe y así sucesivamente, todo se clonará hasta llegar a alguno que no posea jefe (sería el más alto en la jerarquía).

Esquemáticamente se pretende que el grafo quede de la siguiente manera:



En el ejemplo **EJ0008** se puede observar como se ha modificado la implementación del método **Clone()** que propone la interface **ICloneable**. En primer lugar se realiza una clonación superficial y la misma es apuntada por una variable (**e**) para no perder la referencia. Luego se indaga sobre si **e.Jefe** es distinto de **null**. Si es falso significa que no posee referencia a ningún otro objeto, o sea, esta instancia no posee jefe. En caso de retornar verdadero, esta instancia si posee jefe y se procede a clonarla con la instrucción:

```
e.Jefe = (Empleado)this.Jefe.Clone();
```

Esta línea de código invoca al método **Clone()**. También esta línea de código se encuentra dentro del método **Clone()**, lo que genera una recursividad hasta que se produzca una respuesta de falso a la pregunta **e.Jefe != null** y se finalice la recursividad para proceder a retornar el objeto raíz **e**.

```

public class Empleado : ICloneable
{
    15 referencias
    public string Nombre { get; set; }
    11 referencias
    public Empleado Jefe { get; set; }

    2 referencias
    public object Clone()
    {
        Empleado e =(Empleado)this.MemberwiseClone();
        if (e.Jefe != null) { e.Jefe = (Empleado)this.Jefe.Clone(); }
        return e;
    }
}
  
```

Ej0008

Lo que se podrá observar al ejecutar el código del ejemplo **EJ0008** es que se podrá cambiar el nombre de **E3** y su jefe sin afectar el nombre de **E2** ni el nombre del jefe de **E2**.

2. DELEGADOS

Son objetos

Se usan para derivar métodos

Los **delegados** son objetos que hacen referencia a métodos. Se podrían visualizar como un puntero a una función, aunque estrictamente hablando esto último no es lo más preciso. Los delegados son muy útiles para darle al programador mayor poder cuando desea derivar dinámicamente ejecuciones de piezas de código.

Los **delegados** se pueden utilizar para muchas cosas, veremos algunos ejemplos. Cuando se crea una instancia de un **delegado**, se puede asociar con cualquier método que posea una **firma** compatible y un tipo de devolución. Luego, se puede invocar (o llamar) el método a través de la instancia de delegado.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados. La siguiente línea de código muestra la declaración de un delegado:

```
public delegate int Calculo(int x, int y);
```

En el ejemplo **EJ0009**, se puede observar la declaración de un delegado, su instanciación y la llamada al delegado que provocará la llamada al método.

```
// Declaración del Delegado
public delegate string Delegado(String pTexto);
// Declaración de la variable D del tipo Delegado (Delegado es un delegate)
Delegado D;
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    // Intanciación. Se denomina así al hecho de asignarle
    D = UsoDelDelegado;
}
1 referencia
public string UsoDelDelegado(string pTexto) { return pTexto; }
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    // llamada al Delegado.
    MessageBox.Show(D(Interaction.InputBox("Ingrese un Texto: ")));
}
```

EJ0009

Los tipos de delegado se derivan de la clase **delegate**. Los tipos de delegado están sellados, esto implica que no se pueden derivar de ellos (no se puede

heredar de ellos). Como el delegado instanciado es un objeto, puede pasarse como un parámetro o asignarse a una propiedad. Esto permite que un método acepte un delegado como parámetro y llame al delegado en algún momento posterior dentro de su implementación.

El ejemplo **EJ0010** muestra como una función recibe por parámetro un tipo delegado.

```
// Declaración del Delegado
public delegate void Delegado(String pTexto);
// Declaración de la variable D del tipo Delegado (Delegado es un delegado)
Delegado D;
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    // Intanciación. Se denomina así al hecho de asignarle
    D = UsoDelDelegado;
}
1 referencia
1 referencia
public void UsoDelDelegado(string pTexto) { MessageBox.Show(pTexto); }
1 referencia
public void RecibeDelegado(string pS, Delegado pD) { pD(pS); }
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    RecibeDelegado(Interaction.InputBox("Ingrese Texto: "), D);
}
```

EJ0010

El método **RecibeDelegado** posee dos parámetros, **pS** de tipo **string** y **pD** de tipo **Delegado**. Luego en su implementación realiza: **pD(pS)**.

Lo que está haciendo es usar el delegado y pasarle el string como parámetro. La ventaja se hace evidente cuando visualizamos la posibilidad de pasar un delegado por parámetro, lo que es equivalente a pasar una función por parámetro.

Otra característica de los delegados es que pueden llamar a más de un método cuando se invocan. Esto se conoce como multicasting. Para agregar un método adicional a la lista de métodos del delegado, o sea la lista de invocación, simplemente se requiere agregar los métodos utilizando los operadores '+' o '+='.

El ejemplo **EJ0011** muestra como se realiza esto.

```

// Declaración del Delegado
public delegate void Delegado(String pTexto);
// Declaración de la variable D del tipo Delegado (Delegado es un delegate)
Delegado D1,D2,D3,TodosLosMetodos;
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    // Intanciación. Se denomina así al hecho de asignarle
    D1 = Metodo1; D2 = Metodo2; D3 = Metodo3;
    TodosLosMetodos = D1 + D2;
    TodosLosMetodos += D3;
}
3 referencias
public void Metodo1(string pTexto) { MessageBox.Show("EL método 1 se ejecutó y muestra el texto: " + pTexto); }
3 referencias
public void Metodo2(string pTexto) { MessageBox.Show("EL método 2 se ejecutó y muestra el texto: " + pTexto); }
3 referencias
public void Metodo3(string pTexto) { MessageBox.Show("EL método 3 se ejecutó y muestra el texto: " + pTexto); }
1 referencia

```

EJ0011

Cuando se invoque al delegado **TodosLosMetodos** y se le pase un parámetro de tipo **string** se ejecutarán los métodos: **Metodo1**, **Metodo2** y **Metodos3**.

Si se desea quitar algún método del multicast, por ejemplos el **Metodo1**, se puede realizar de la siguiente manera.

TodosLosMetodos -= Metodo1;

Además, a los delegados se le pueden pasar funciones lambda o bien aprovecharlos de maneras muy versátiles. El ejemplo **EJ0012** muestra como hacerlo.

```

// Declaración del delegado.
delegate bool Dele(string pString);
delegate IEnumerable<string> Dele2(string[] pArray, int plimite);

3 referencia
private void Form1_Load(object sender, EventArgs e)...
3 referencia
IEnumerable<string> FuncPrueba(string[] pArray,int plimite)
{
    return pArray.Where(n => n.Length >= plimite);
}
3 referencia
private void button1_Click(object sender, EventArgs e)
{
    Dele ejemplodelegado = pCadena => { return pCadena.Length > 6; };
    MessageBox.Show("La cadena ingresada mide más de 6 caracteres: " +
        ejemplodelegado(Interaction.InputBox("Ingresa el Texto:")));
}
3 referencia
private void button2_Click(object sender, EventArgs e)
{
    string[] Nombres = {"Ana", "Pedro", "Cecilia", "Guillermo", "Sol", "Ezequiel",
        "Martin", "María", "Marcelo", "Adriana" };
    Dele2 ejemplodelegado2 = FuncPrueba;
    textBox1.Clear();
    foreach (string S in ejemplodelegado2(Nombres,
        int.Parse(Interaction.InputBox("Ingresa la cantidad de caracteres mínima"))))
    {
        textBox1.Text += S + Environment.NewLine;
    }
}

```

EJ0012

Se observan dos delegados **Dele** y **Dele2**. El primero define que se recibe un string y retorna un bool.

El segundo define que recibe un array de string como primer parámetro y un entero como segundo. Retorna un IEnumerable<string>.

En el ejemplo, los parámetros de entrada de **Dele2** sirven para: el primero para recibir un array de nombres y el segundo para recibir un número que determina la cantidad mínima de caracteres que debe tener el nombre para ser seleccionado.

Luego a la variable **ejemplodelegado** de tipo **Dele** se le asigna una expresión lambda. Esta función anónima determina si la cantidad de caracteres de la palabra ingresada posee más de 6 caracteres y retorna un valor bool en base a esto.

La variable **ejemplodelegado2** de tipo **Dele2**, delega o apunta a la función **FuncPrueba**. La misma mediante LinQ retorna un IEnumerable con los nombres del array, cuando poseen una cantidad igual o superior a límite de caracteres que se ingresaron. Luego este resultado se recorre con un foreach para mostrarlo en una caja de texto.



Guía de Revisión Conceptual

1. ¿Para qué se utilizan las interfaces?
2. ¿Cómo se implementa una interfaz?
3. ¿Se pueden heredar las interfaces?
4. ¿Se puede implementar un tipo de polimorfismo peculiar por medio de interfaces?
5. ¿Para qué se utiliza la interfaz *Comparable*?
6. ¿Para qué se utiliza la interfaz *Comparable*?
7. ¿Para qué se utiliza la interfaz *Cloneable*?
8. ¿Para qué se utiliza la interfaz *Enumerable*?
9. ¿Para qué se utiliza la interfaz *IEnumerator*?
10. ¿Qué es un delegado?
11. ¿A qué elementos se les puede delegar?
12. ¿Qué se puede delegar?
13. ¿Cómo construiría un delegado?
14. ¿Cómo implementaría un procedimiento de devolución de llamadas?
15. ¿Para qué sirve la multidifusión de delegados?

Intente desarrollar los siguientes ejercicios y compártalos en el foro con sus compañeros y tutor.

1. Desarrolle un programa que implemente la interfaz *Comparable*. Genere un *array* de objetos y ordénelos.

2. Desarrolle un programa que implemente la interfaz *IComparer*. La clase que lo implementa deberá tener al menos cinco criterios de ordenamiento. Genere un *array* de objetos y ordénelos por cada criterio.
3. Desarrolle un programa que implemente la interfaz *IClonable*. Clone el objeto que posee la interfaz y demuestre esto utilizando “*IS*”.
4. Desarrolle un programa que implemente las interfaces *IEnumerable* e *IEnumerator*. Lo adaptado recórralo con el *for each* y muestre los resultados.

Cierre de la unidad

Esperamos que le haya gustado esta unidad y haya podido apropiarse de nuevos conocimientos para desarrollar software más robusto.

Le proponemos que revise toda la bibliografía y responda las preguntas a fin de reforzar los conceptos. Si posee dudas contáctese con el tutor y agótelas.



Tenga en cuenta que los trabajos que produzca durante los procesos de estudio son insumos muy valiosos y de preparación para la Evaluaciones Parciales. Por lo tanto, guarde sus notas, apuntes y gráficos, le serán de utilidad.