Tutorial de C# Programación OO

Por: Óscar López, M.Sc.

olopez@uniandino.com.co

¡Hola Mundo!

```
1.    using System;

2.    namespace HolaMundo {

3.      /// <summary>
4.      /// Summary description for Hola.
5.      /// </summary>
6.      class Hola {

7.        /// <summary>
8.        /// The main entry point for the application.
9.        /// </summary>
10.       [STAThread]
11.       static void Main() {
12.         Console.WriteLine("¡Hola Mundo!");
13.       }
14.
15.     }

16.   }
```

CREACIÓN DE OBJETOS

Clases - Roles y Funciones

- Generador de nuevos objetos
- Descripción de la representación de sus instancias
- Descripción del protocolo de mensajes de sus instancias
- Elemento de la descripción de la taxonomía de un objeto
- Un medio para implementar programación diferencial
- Repositorio de métodos para recibir mensajes
- Artefacto para actualizar muchos objetos cuando ocurre un cambio
- Conjunto de todas las instancias de una clase

Clases - Definición

```
[[atributos]] [modificadores]
class identificador [: base,]
{cuerpo clase}
```

- *Atributos:* anotaciones usadas para proporcionar meta-datos sobre la clase
- *Identificador:* nombre, notación de Pascal
- *Base:* Clase, Clase Abstracta, Interfaz(ces). Atención: sólo una clase, y las interfaces van al final

Modificadores de Clase

Modificador	Comentario
`new`	Sólo aplica para clases anidadas, las “esconde”
`abstract`	Declara la clase como *abstracta*
`internal`	Declara una clase anidada
`sealed`	Declara la clase como *sellada*

Modificadores de Acceso

Modificador	Comentario
`public`	Sin restricciones. Visible desde cualquier método de cualquier clase
`private`	Sólo es visible desde los métodos de la misma clase
`protected`	Visible desde los métodos de una clase y todas sus sub-clases
`internal`	Visible desde los métodos de todas las clases dentro de un mismo ensamblaje
`protected internal`	Visible desde los métodos de una clase, sus sub-clases, y todas las clases en su mismo ensamblaje

Aplicabilidad de Modificadores de Acceso

Categoría	Modificadores Aplicables	Acceso por Defecto
espacios de nombres	(implícito) `public`	`public`
clases, interfaces, estructuras	`public, internal`	`internal`
miembros de clase (incluyendo tipos anidados)	`public, protected, internal, protected internal, private`	`private`
miembros de estructura	`public, internal, private`	`private`
miembros de interfaz	(implícito) `public`	`public`
miembros de enumeración	(implícito) `public`	`public`

Instanciación - Palabras Clave

new

crea nuevas instancias de una clase o una estructura

this

referencia a la instancia actual de un objeto

base

referencia a la super-clase de un objeto

Constructores de Instancia

- El nombre del constructor debe ser el mismo del nombre de la clase
- No tienen un tipo de retorno
- Si no se especifica, se crea un constructor por defecto
- Los constructores se pueden sobrecargar y pueden llamar a otros constructores dentro de la misma clase o en una super-clase

Constructores de Instancia - Ejemplo

```
1.    public class Window {

2.      int top, left;
3.
4.      public Window(int top, int left) {
5.        this.top = top;
6.        this.left = left;
7.      }
8.    }

9.    public class ListBox : Window {

10.     string mListBoxContents;

11.     ListBox(int top, int left, string theContents)
12.     : base(top, left) {
13.       mListBoxContents = theContents;
14.     }

15.     ListBox(int top, int left)
16.     : this(top, left, "") {
17.     }
18.   }
```

Constructores Estáticos

- No puede tener parámetros
- No puede tener modificadores de acceso
- No puede ser llamado explícitamente
- Es llamado una sola vez, antes de crear la primera instancia

```
1.   public class Window {

2.     static GLibrary gLib;

3.       static Window() {
4.         gLib = getLib();
5.       }
6.   }
```

Inicializadores, Valores por Defecto

```
int num = -1;
```

Tipo	Valor por Defecto
Numérico (`int`, `long`, etc.)	`0`
`bool`	`false`
`char`	`'\0'`
`enum`	`0`
referencia	`null`

Destructores

- ¡Recolector de Basura!
- Normalmente no son necesarios, sólo si se deben liberar recursos “costosos”
- Los destructores se llaman cuando un objeto es recolectado, es un proceso no-determinístico
- No se pueden llamar explícitamente (pero un objeto se puede “marcar” para ser recolectado asignándole `null`)

```
1.  ~Window() {
2.    gLib.release();
3.  }
```

```
1. protected override void Finalize() {
2.   try {
3.     gLib.release();
4.   }
5.   finally {
6.     base.Finalize();
7.   }
8. }
```

Implementando IDispose

```
1.      using System;
2.
3.      class TestDispose : IDisposable {

4.        bool isDisposed = false;

5.        protected virtual void MyDispose() {
6.          if ( !isDisposed ) { // sólo limpiar una vez!
7.            Console.WriteLine("Limpiando..."); // realizar la limpieza
8.          }
9.             this.isDisposed = true;
10.       }

11.       public void Dispose() {
12.         MyDispose();
13.         // decirle al recolector que no debe llamar al destructor
14.            GC.SuppressFinalize(this);
15.       }

16.       ~TestDispose() {
17.         MyDispose();
18.       }
19.     }
```

Declaración *using*

- Debido a que no hay seguridad de que Dispose() sea llamado, C# proporciona la declaración *using,* la cual garantiza que Dispose() sea llamado automáticamente cuando se llega al corchete que cierra la declaración

```
1.  Font theFont = new Font("Arial", 10.0f);
2.  using (theFont) {
3.    // usar theFont
4.  } // el compilador invocará Dispose() en theFont
```

Clases Anidadas

- Típicamente, existe sólo para servir a la clase que la contiene, como una clase auxiliar
- Puede ocultarse de otras clases si se declara como privada
- Una clase anidada tiene acceso a todos los miembros de la clase que la contiene, aún los privados
- Puede acceder a todos los miembros externos que puedan ser accedidos por la clase que la contiene

Clases Anidadas - Ejemplo

```
1.    public class Fraction {
2.      internal class FractionPaint {
3.        public void Draw(Fraction f) {
4.          Console.WriteLine("Drawing Fraction…");
5.        }
6.      }
7.    }
```

```
Fraction f1 = new Fraction(3,4);
Fraction.FractionArtist fp = new Fraction.FractionPaint();
fp.Draw(f1);
```

Estructuras

```
[[atributos]] [modificadores acceso]
struct identificador [: interfaz,]
{cuerpo estructura}
```

- Otro tipo de datos definido por el usuario
- Una alternativa “liviana” a las clases (menos memoria, menos funcionalidad)
- Pueden tener: constructores, propiedades, métodos, campos, operadores, tipos anidados y propiedades indexadas; pueden implementar interfaces; se les reserva memoria al declararlas
- No soportan destructores ni herencia, no pueden tener un constructor sin parámetros ni usar inicializadores; sus “instancias” son de tipo valor (a diferencia de las instancias de una clase, que son de tipo referencia)

Estructuras - ¿Para Qué?

- Deberían usarse para tipos pequeños, simples, con comportamiento y características similares a las de los tipos básicos. Ej.: un Punto con coordenadas (x, y)
- Las operaciones sobre estructuras pueden ser más rápidas en algunos casos, pero las asignaciones son más costosas
- Encapsular los miembros haciéndolos privados y accediéndolos usando propiedades o métodos es exagerar … en este caso acceder directamente al estado de una estructura es más razonable
- Es más eficiente usar estructuras en Arreglos que en Colecciones

Estructuras - Ejemplo

```
1.    using System;

2.    public struct Point {

3.      int x, y;

4.        public Point(int xCoordinate, int yCoordinate) {
5.          x = xCoordinate;
6.          y = yCoordinate;
7.        }

8.    }

9.    public class Tester {
10.     static void Main() {
11.       Point p1 = new Point(300,200);
12.       Point p2;
13.       p2.x = 640;
14.       p2.y = 480;  // No olvidar asignar todos los campos
15.     }
16.   }
```

COMPARTIR INFORMACIÓN

Herencia

- En C#, una clase sólo puede tener *una* super-clase, y todas las clases heredan de `System.Object`
- Una sub-clase hereda todos los miembros *visibles* de su super-clase, y son tratados como si hubieran sido definidos en la sub-clase
- Las siguientes clases *no* pueden ser extendidas:
  - `System.Array`
  - `System.Delegate`
  - `System.Enum`
  - `System.ValueType`

Métodos de `System.Object`

Método	Comentario
`public virtual string` **`ToString`**`()`	representación textual del objeto
`public virtual int` **`GetHashCode`**`()`	*hash* del objeto
`public virtual bool` **`Equals`**`(object a)`	prueba igualdad entre objetos
`public static bool` **`Equals`**`(object a,object b)`	prueba valores nulos, luego igualdad
`public static bool` **`ReferenceEquals`**`(object a,object b)`	prueba identidad
`public Type` **`GetType`**`()`	instancia de `Type` del objeto
`protected object` **`MemberwiseClone`**`()`	copia superficial
`protected virtual` **`Finalize`**`()`	limpieza

Polimorfismo

- Capacidad de usar “muchas formas” de un mismo tipo
- Un miembro debe ser declarado explícitamente como `virtual` para poder declarar `override` sobre éste en una sub-clase
- Un miembro `virtual` no puede ser declarado como: `static`, `abstract`, `override` o `private`
- Sólo los siguientes miembros pueden ser declarados como `virtual`: métodos, propiedades y propiedades indexadas

Uso de `virtual` y `override`

- Una sub-clase no puede hacer *override* de los miembros privados de una super-clase
- Si una sub-clase tiene un método con la misma firma de un método privado en una super-clase, no se considera un *override*
- El que un miembro de una clase sea declarado como virtual no obliga a que sus (posibles) sub-clases deban hacer *override* de éste

Escondiendo Miembros con new

new

como modificador, al crear un nuevo miembro de clase (método, campo, constante, propiedad, tipo) que “esconde” el miembro heredado de una super-clase

- Una sub-clase es libre de implementar su propia versión de un miembro de una super-clase. La palabra clave `new` indica que la sub-clase, intencionalmente, ha escondido y reemplazado un miembro de su super-clase

Escondiendo Miembros - Ejemplo

```
1.    using System;

2.    class A {
3.      public virtual void DoThis() {
4.        Console.WriteLine("A");
5.      }
6.    }

7.    class B : A { // B extiende A
8.      public new void DoThis() {
9.        Console.WriteLine("B");
10.     }
11.   }

12.   class Test {
13.     public static void Main() {
14.      A temp1 = new A();
15.      temp1.DoThis();
16.      B temp2 = new B();
17.      temp2.DoThis();
18.      A temp3 = new B();
19.      temp3.DoThis();
20.      A temp4 = (A)temp2;
21.      temp4.DoThis();
22.     }
23.   }
```

Cuando se usa `new`, el tipo de dato de la variable usada para referenciar un objeto se toma en cuenta a la hora de determinar cuál de los dos métodos invocar. Si la variable es de tipo A, no importa si la instancia a la que se refiere es de clase A o B, se llama el método DoThis() en A. No se trata de un *override* de B, simplemente, se lo ha "escondido"

Clases Abstractas

- Obligan a que las subclases implementen uno o más métodos
- No se pueden instanciar
- Si una clase tiene un solo método abstracto, ésta también debe declararse como abstracta
- Se antepone la palabra clave `abstract` a la definición de un método, no se implementa
- Típicamente, se encuentran en la parte superior de una jerarquía de clases

Clases Selladas

- Lo opuesto a una clase abstracta
- No permite que una clase tenga sub-clases
- Se antepone la palabra clave `sealed` a la definición de una clase
- Se marcan como tal para prevenir “herencia accidental”
- Se debería marcar como sellada una clase en donde todos sus miembros son estáticos

Interfaces

```
[[atributos]] [modificadores acceso]
interface identificador [: interfaz,]
{cuerpo interfaz}
```

- Un contrato, un conjunto de capacidades ofrecidas por un objeto
- En C#, una interfaz puede contener métodos, propiedades, propiedades indexadas y eventos; *no* campos
- Debe detallar el tipo de retorno, el nombre y los parámetros de un método, sin especificar modificadores de acceso
- Una interfaz no puede ser instanciada
- Una clase puede *implementar* una o más interfaces
- Por convención, su nombre empieza por ‘I’

Conversiones de Interfaces - `is`, `as`

- `IStorable isDoc = (IStorable) doc;`

- ```
  if (doc is IStorable)
        IStorable isDoc = (IStorable) doc;
  ```

- `IStorable isDoc = doc as IStorable`
  ```

¿Qué Define a un Objeto?

- Estado
- Comportamiento
- Identidad
- Eventos

Posibles Miembros de una Clase

- Estado
  - Campos
  - Constantes
  - Campos de sólo lectura
  - Enumeraciones
- Comportamiento
  - Constructores de instancia
  - Constructores estáticos
  - Destructores
  - Tipos anidados
  - Propiedades
  - Propiedades Indexadas
  - Métodos
  - Operadores
  - Delegados y Eventos

Miembros de Clase y de Instancia

- Los siguientes miembros pueden ser declarados como estáticos, usando la palabra clave `static`: constructores, campos, campos de sólo lectura, propiedades, métodos, operadores, eventos
- Los miembros de instancia están asociados con instancias de un tipo
- Los miembros estáticos están asociados con una clase y no con una instancia en particular y como tal, se acceden anteponiendo el nombre de la clase y el operador '.' al nombre del miembro
- Un miembro estático no puede ser declarado con ninguno de éstos modificadores: `virtual`, `override`, `abstract`

ESTADO

Campos

- Variable asociada a una instancia o a una clase, almacena los datos que definen las características propias de un objeto. No confundir con atributos ni con propiedades
- Por convención, todos los campos son privados. Si se necesita proporcionar acceso a éstos, se recomienda usar propiedades
- También por convención, se recomienda usar la convención “Camel” para los nombres de los campos (primera letra en minúsculas, palabras separadas por una letra mayúscula)
- El nombre de un campo no puede ser una palabra reservada

Constantes

- Representan valores que no cambian durante la ejecución del programa
- Se declaran usando la palabra clave `const`
- Pueden ser de tipos valor (`sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char`) y `string`
- Pueden ser variables o campos de instancia
- Son evaluadas en tiempo de compilación

```
// Número de milisegundos en un día
const long MS_POR_DIA = 1000*60*60*24;
```

Campos de Sólo Lectura

- Se declaran usando la palabra clave `readonly`
- Son más generales que las constantes
- Pueden ser de cualquier tipo que sea evaluado una y sólo una vez en tiempo de ejecución
- Típicamente, son inicializados al momento de cargar una clase (en el caso de campos estáticos) o al momento de inicializar una instancia (para los campos de instancia)
- No sólo se pueden aplicar a campos constantes: se pueden usar para datos que, una vez son asignados, son invariantes (Ej., una cédula)
- No confundir: un campo de sólo lectura *no* es inmutable; cuando se aplica a un tipo referencia, sólo impide que la referencia al objeto sea cambiada

Campos de Sólo Lectura - Ejemplo

```
1.  using System;
2.  using System.Threading;

3.  public class ReadOnly {

4.    int id;
5.    readonly DateTime timeOfInstanceCreation = DateTime.Now;
6.    static readonly DateTime timeOfClassLoad = DateTime.Now;
7.    static readonly ReadOnly ro = new ReadOnly();

8.    public ReadOnly() {
9.      Console.WriteLine("Clase cargada: {0}, Instancia creada: {1}",
10.                       timeOfClassLoad, timeOfInstanceCreation);
11.   }

12.   public static void Main() {

13.     const int size = 10;
14.     for (int i = 0; i < size; i++) {
15.       new ReadOnly();
16.       Thread.Sleep(1000);
17.     }

18.     // Se puede cambiar un miembro
19.     ro.id = 5;
20.     Console.WriteLine(ro.id);
21.     Console.ReadLine();

22.     // El compilador dice "a static readonly field cannot be assigned to"
23.     // ro = new ReadOnly();

24.   }

25. }
```

Enumeraciones

```
[[atributos]] [modificadores]
enum identificador [: base]
{lista de enumeración,}
```

- Proporcionan una alternativa al uso de constantes
- Permiten agrupar constante relacionadas lógicamente
- Son un tipo valor; consisten en un conjunto de constantes con nombre (denominadas la lista de enumeración)
- La base es el tipo usado “por debajo” para representar la enumeración. Puede ser cualquiera de los tipos enteros excepto `char`
- Si no se especifica una base, por defecto se utiliza `int`

Enumeraciones

- Cada elemento de una enumeración corresponde a un valor numérico. Por defecto, una enumeración empieza en 0 y cada valor siguiente aumenta una unidad respecto al anterior

```
1.  enum algunosValores {
2.    Primero,       //  0
3.    Segundo,       //  1
4.    Tercero = 20, // 20
5.    Cuarto         // 21
6.  }
```

Enumeraciones - Ejemplo

```
1.  using System;

2.  class Values {

3.    // declaración de la enumeración

4.    enum Temperatures : int {
5.      WickedCold = 0,
6.      FreezingPoint = 32,
7.      LightJacketWeather = 60,
8.      SwimmingWeather = 72,
9.      BoilingPoint = 212
10.   }

11.   static void Main() {
12.     Console.WriteLine("Temperatura de congelación del agua: {0}",
13.                       (int) Temperatures.FreezingPoint);
14.     Console.WriteLine("Temperatura de ebullición del agua: {0}",
15.                       Temperatures.BoilingPoint);
16.     Console.ReadLine();
17.   }

18. }
```

COMPORTAMIENTO

Propiedades

```
[modificadores] tipo identificador {
   get {
        return campo;
   }
   set {
        campo = value;
   }
}
```

- Una forma muy conveniente para encapsular un campo
- Permite acceder a los campos usando métodos en forma transparente
- Recomendado seguir la convención de Pascal para los identificadores
- `get` o `set` son opcionales: propiedades de sólo lectura o de sólo escritura
- No necesariamente debe existir un campo para cada propiedad: valores derivados
- Pueden heredarse de manera selectiva, declarándolas como `virtual`

Propiedades - Ejemplo

```
1.  using System;

2.  public class TestClass {
3.    private string color = “yellow”;
4.    public string Color {  // propiedad Color
5.      get {
6.        return color;
7.      }
8.      set {
9.        color = value;
10.     }
11.   }
12.   public static void Main() {
13.     TestClass c = new TestClass();
14.     Console.WriteLine(c.Color); // get
15.     c.Color = “blue”;           // set
16.     Console.WriteLine(c.Color); // get
17.   }
18. }
```

Propiedades Indexadas

```
[modificadores] tipo this [tipo argumento,] {
   get {
        ;
   }
   set {
        ;
   }
}
```

- Una forma conveniente para tratar un objeto que encapsula un arreglo o una colección como si el objeto mismo fuera un arreglo
- Una propiedad indexada se usa para representar la lista de objetos contenidos en una clase, usando la notación `objeto[índice]`
- Puede pasarse más de un argumento entre [], y de diferentes tipos
- También puede ser declarado como `virtual`
- Se recomienda no poner más de una propiedad indexada por clase

Propiedades Indexadas - Ejemplo

```
1.  using System;

2.  class TestClass {
3.    // este es el arreglo encapsulado
4.    string[] myArray = new string[10];

5.    // constructor
6.    public TestClass() {
7.      for (int i=0; i<10; i++)
8.        myArray[i] = "hola";
9.    }

10.   // indexamos usando un entero
11.   public string this[int index] {
12.     get {
13.       return myArray[index];
14.     }
15.     set {
16.       myArray[index] = value;
17.     }
18.   }

19.   // indexador sobrecargado usamos una cadena
20.   public int this[string s] {
21.     get {
22.       for (int i=0; i<10; i++) {
23.         if (myArray[i].Equals(s))
24.           return i;
25.       }
26.       return -1;
27.     }
28.   }
```

```
29.   public static void Main() {

30.     TestClass c = new TestClass();

31.     // Asignamos valores a myArray
32.     c[2] = "mundo";
33.     c[3] = "mundo";
34.     c[5] = "mundo";

35.     // indexador sobrecargado
36.     Console.WriteLine("indice que " +
37.       "contiene mundo es " + c["mundo"]);

38.   }
39. }
```

Métodos

```
[[atributos]][modificadores]
tipo nombre ([parámetros]) {cuerpo}
```

- Se recomienda usar la convención de nombres de Pascal
- Si lo que se desea es un procedimiento, el tipo de retorno es `void`
- Un método que hace *override* puede ser declarado como `sealed`
- La firma de un método está compuesta de su nombre y su lista de parámetros
- Un método se puede sobrecargar variando el número o el tipo de sus parámetros

Modificadores Válidos de Métodos

- `new`
- `abstract`
- `static`
- `override`
- `virtual`
- `sealed`
- Modificadores de acceso

Uso de `params`

- Permite utilizar arreglos unidimensionales como listas de parámetros de tamaño variable
- No forma parte de la firma de un método
- Sólo puede usarse una por método
- Si un método tiene más de un parámetro, `params` debe declararse de último
- Si lo que se desea es una lista de tamaño *y* tipo variable, utilizar `params object []`

Uso de params – Ejemplo

```
1.  using System;

2.  class Test {

3.    public void DisplayVals(params int[] intVals) {
4.      for (int i = 0; i < intVals.Length; i++) {
5.        Console.WriteLine("DisplayVals {0}", intVals[i]);
6.      }
7.    }

8.    static void Main() {
9.      Test t = new Test();
10.     t.DisplayVals(1,2,3,4,5);
11.     int [] explicitArray = new int[5] {6,7,8,9,10};
12.     t.DisplayVals(explicitArray);
13.     Console.ReadLine();
14.   }
15. }
```

Parámetros por Referencia

- Los tipos referencia siempre se pasan por referencia
- Por defecto, los tipos valor siempre se pasan por valor
- Se puede usar el modificador de parámetro `ref` para pasar tipos valor por referencia
- También, se puede usar el modificador de parámetro `out` para los casos en los que se desea pasar un parámetro por referencia sin tener que inicializarlo primero
- Si un parámetro es declarado `ref`, debe asignársele un valor antes del método. Si es declarado `out`, debe asignársele un valor dentro del método
- Típicamente, se utiliza para “retornar” valores múltiples desde un método

Parámetros por Referencia - Ejemplo

```
1.  public class Test {
2.    public static void Main() {
3.      int a = 0; // a es inicializado
4.      Test.DoSomething(ref a);
5.      System.Console.WriteLine(a);
6.    }
7.    static void DoSomething(ref int y) {
8.      y = 42;
9.    }
10. }
```

```
1.  public class Test {
2.    public static void Main() {
3.      int a;  // a no es inicializado
4.      Test.DoSomething(out a);
5.      System.Console.WriteLine(a);
6.    }
7.    static void DoSomething(out int y) {
8.      y = 42;
9.    }
10. }
```

ORGANIZACIÓN

Espacios de Nombres

- Una forma de organizar elementos e identificarlos de manera única
- Evita el riesgo de colisiones de nombres
- Dos espacios de nombres pueden contener la misma clase
- Pueden anidarse
- Dentro de éstos pueden definirse alias

Espacios de Nombres - Ejemplo

```
1.  namespace TutorialCSharp {
2.    namespace TutorialCSharpTest {
3.      using System;
4.      using C = System.Console; // definición de alias
5.      public class Tester {
6.        public static int Main() {
7.          for (int i=0;i<10;i++)
8.            C.WriteLine("i: {0}",i); // uso de alias
9.          return 0;
10.       }
11.     }
12.   }
13. }
```

- El nombre completo de la clase Tester es:

```
TutorialCSharp.TutorialCSharpTest.Tester
```

Ensamblajes

- Son la unidad básica de reutilización, versiones, seguridad y despliegue en .NET
- Se trata de una colección de archivos que aparecen como un sólo archivo .dll o .exe
- Además de tener una colección de clases y métodos, también contienen otros recursos como archivos con imágenes, definiciones de tipos para cada clase definida y meta-datos sobre el código y los datos
- Consisten de uno o más módulos, con un sólo punto de entrada: `DLLMain`, `WinMain` o `Main`
- Se cargan sobre demanda, y nunca son cargados si no se necesitan

Ensamblajes

Meta-datos

Información almacenada que especifica los tipos y métodos del ensamblaje, describiendo por completo el contenido de cada módulo: los ensamblajes son auto-descriptivos

Seguridad

Forman un límite de seguridad al limitar el alcance de los tipos que contienen: la definición de un tipo no puede cruzar dos ensamblajes

Versiones

Tienen un número de versión que se refiere al contenido de un sólo ensamblaje. Todos los tipos y recursos dentro del ensamblaje cambian de versión juntos. Son de la forma `mayor:menor:construcción:revisión`

Ensamblajes - Manifiestos

- Como parte de sus meta-datos, cada ensamblaje tiene un manifiesto
- Describe qué hay en el ensamblaje: información de identificación (nombre, versión, etc.), una lista de tipos y recursos, un mapa que conecta los tipos públicos con el código que los implementa, y una lista de los ensamblajes referenciados por este ensamblaje
- Pueden examinarse ejecutando un des-ensamblador, (Ej. `ildasm`) sobre un ensamblaje
- Incluye un código de *hash* que identifica cada módulo y garantiza que cuando un programa se ejecute, sólo se cargue la versión adecuada de un módulo
- A su vez, cada módulo tiene su propio manifiesto, separado del manifiesto del ensamblaje
- Las referencias a otros ensamblajes incluyen el nombre, la versión, la cultura y opcionalmente el “originador”: firma digital del desarrollador o compañía que proporciona el ensamblaje

Ensamblajes - Multi-Módulo

- Un ensamblaje de un sólo módulo consiste de un sólo archivo .dll o .exe que contiene todos los tipos e implementaciones de la aplicación, junto con su manifiesto
- Un ensamblaje multi-módulo consiste de múltiples archivos: 0 ó 1 .exe y 0 ó más .dll; aunque debe tener al menos un .exe o .dll. El manifiesto reside en un archivo aparte o puede estar en alguno de los módulos
- Cuando el ensamblaje es referenciado, el entorno de ejecución carga el archivo con el manifiesto y carga los módulos a medida que vaya siendo necesario
- Útil cuando se tiene muchos desarrolladores trabajando en distintas partes de un mismo programa
- Se crean manipulando los archivos `assemblyInfo.cs`, `makefile` en conjunto con las herramientas `nmake` y `csc`

Ensamblajes - Privados

- Diseñados para ser usados por una sola aplicación
- Por defecto, los ensamblajes son privados
- Los archivos son mantenidos en un mismo directorio (y sus subdirectorios), nada más depende de ellos y se pueden “instalar” en otra máquina copiándolos directamente
- Puede tener cualquier nombre, puesto que es un nombre local

Ensamblajes - Compartidos

- Diseñados para ser compartidos entre muchas aplicaciones
- Útil cuando se tienen componentes genéricos que van a ser usados por varios programas
- Deben cumplir tres condiciones:
  - Tener un nombre “fuerte” : una firma digital usando `sn`; garantiza que sea globalmente único, permite verificar al desarrollador y detectar alteraciones
  - Tener información sobre su versión: Los ensamblajes son identificados por su nombre *y* su versión; varias versiones de un mismo ensamblaje pueden convivir
  - Situarse en el Caché de Ensamblajes Global (GAC), un área del sistema de archivos reservada para almacenar ensamblajes compartidos