

# Tipos de datos en .NET - Parte 1. C#

Por Mauro Sant'Anna ([mas\\_mauro@hotmail.com](mailto:mas_mauro@hotmail.com)). Mauro es Director regional de MSDN, consultor e instructor de MAS Informática ([www.mas.com.br](http://www.mas.com.br)) y lleva impartiendo formación acerca de la arquitectura .NET desde octubre de 2000.

Todos los lenguajes de la arquitectura .NET tienen muchas cosas en común. Una de ellas es el conjunto de tipos disponibles. El sistema de tipos .NET es muy característico y sustancialmente diferente a otros lenguajes como C++ y Visual Basic. En esta serie de dos artículos, exploraré el sistema de tipos y, en especial, sus características únicas. Utilizaré la nomenclatura y ejemplos en C#, sin embargo los tipos están también disponibles en otros lenguajes como Visual Basic.

Dividiré los tipos en tres categorías: "por valor", "por referencia" y, el tipo "string", que tiene algunas características de ambos. Además de dos tipos intrínsecos, es posible definir nuevos tipos, tanto por valor como por referencia.

## Tipos por valor.

Los tipos por valor tienen las siguientes características principales:

- Se ubican directamente en la pila (stack).
- No necesitan ser iniciados como un operador nuevo.
- La variable almacena el valor directamente.
- La atribución de una variable a otra copia o contenido, creando efectivamente otra copia de la variable.
- Normalmente utilizados con tipos pequeños (menos de 16 bytes), donde el uso de referencias ocasionaría un costo mayor.
- Se pueden convertir automáticamente para referencias en un proceso llamado "boxing", descrito en la segunda parte de este artículo.
- No pueden contener el valor null (cero).

## Tipos por valor.

Tipo	Implementación
byte	Entero de 8 bits sin señal (0 a 255)
sbyte	Entero de 8 bits con señal (-127 a 128)
ushort	Entero de 16 bits sin señal (0 a 65 535)
short	Entero de 16 bits con señal (-32 768 a 32 767)
uint	Entero de 32 bits sin señal (0 a 4 294 967 295)
int	Entero de 32 bits con señal (-2 147 483 648 a 2 147 483 647)
ulong	Entero de 64 bits sin señal (0 a 18 446 744 073 709 551 615)
long	Entero de 64 bits con señal (-9 223 372 036 854 775 808 a 9 223 372 036 854 775 807)
double	Punto flotante binario IEEE de 8 bytes ( $\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$ ), 15 dígitos

	decimales de precisión.
float	Punto flotante binario IEEE de 4 bytes ( $\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$ ), 7 dígitos decimales de precisión.
decimal	Punto flotante decimal de 128 bits. ( $1.0 \times 10^{-28}$ a $7.9 \times 10^{28}$ ), 28 dígitos decimales de precisión.
bool	Puede tener los valores true y false (verdadero y falso). No es compatible con enteros.
char	Un solo carácter Unicode de 16 bits. No es compatible con enteros.

Los tipos enteros vienen en cuatro tamaños: uno, dos, cuatro y ocho bytes, con y sin señal. Los dos tipos de punto flotante IEEE no ofrecen ninguna novedad.

El tipo decimal es una novedad: tiene 28 dígitos de precisión y las cuentas son menos propensas a errores de dejar los números redondeados comunes a los formatos IEEE y odiados por quienes crean software de contabilidad. Por ejemplo, con el decimal 14,2 más 0,2 da 14,4 en vez de 14,39999999999999 que usted obtendría con el tipo double (doble).

El tipo bool almacena los valores true y false. Las expresiones lógicas como if y while esperan siempre una expresión del tipo bool. Los tipos enteros no pueden ser utilizados en estas situaciones, como usted haría en C++.

El tipo char almacena caracteres en el modelo Unicode. Este modelo incluye letras en diversos alfabetos aparte de dos "romanos" utilizados en el occidente. Incluye alfabetos como Cirílico, Árabe, Hebreo, Chino, Japonés, Coreano y Sánscrito.

## **Tipos por valor definidos por el usuario.**

En C# usted puede declarar nuevos tipos, de manera similar al C++ y Pascal.

### **Enum.**

Permite declarar una secuencia de identificadores asociados, pero no compatibles con enteros y con otras enumeraciones.

Ejemplo:

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

### **Struct.**

Permite declarar tipos que contienen varios valores, identificados por un nombre. Similar a struct de C++ y a record de Pascal. Ejemplo:

```
public struct Point {
    public int x, y;
    public Point(int p1, int p2) {
        x = p1;
```

```
        y = p2;  
    }  
}
```

Structs poseen algunas características en común con las clases:

- Pueden tener métodos.
- Pueden tener constructores.

Existen, por lo tanto, diferencias con relación a las clases:

- Estos son tipos por valor en cuanto las clases son tipos por referencia.
- No podemos declarar un constructor que no acepte argumentos.
- Podemos atribuir la variable this.
- No soportan herencia; son implícitamente sealed (cerradas).

Structs suministran una alternativa más "ligera y barata" que las clases, en donde el costo del uso de las clases (reserva dinámica de memoria, métodos virtuales y uso de punteros) sería muy costoso. Por ejemplo, un punto (coordenada X, Y).

## **Tipos por referencia.**

Los tipos por referencia tienen las siguientes características principales:

- Son reservados en un heap y sujetos a la "colección de basura" cuándo no sean más utilizados.
- Deben ser iniciados con acelerador new.
- La variable almacena una "referencia", una especie de puntero; el contenido en si queda en heap.
- La atribución de una variable a otra copia de referencia; podemos tener muchas variables refiriendo al mismo valor.
- Normalmente utilizados con tipos de gran tamaño (más de 16 bytes), en donde el costo de la reserva dinámica es relativamente pequeño comparado con su flexibilidad.
- Pueden contener el valor null; aún si utilizamos una variable con el valor null se generará la excepción NullReferenceException.

## **Object (Objeto)**

**Tipo intrínseco.** Es la clase base de todas las otras clases y tipos. Una variable del tipo object puede contener valores de cualquier tipo. Los tipos por referencia son almacenados directamente; los tipos por valor son blancos de "boxing".

## **Arrays (Arreglos)**

Un array siempre se crea dinámicamente al momento de la ejecución. Podemos tener arrays de varias dimensiones y arrays de arrays. Vea un ejemplo de creación e iniciación de un array de enteros de una dimensión:

```
int[] myIntArray = new int[5] { 1, 2, 3, 4, 5 };  
Class (Clase)
```

Tipo definido por el usuario. Las clases son siempre derivadas de object y pueden contener campos, métodos y propiedades. Una clase se puede derivar de alguna otra clase única, y también de varias interfaces. Vea un ejemplo:

```
public class Tiempo {  
    protected int H; protected int M; protected int S;  
    public Tiempo() {  
        Adjusta(0, 0, 0);  
    }  
    public Tiempo(int _H, int _M, int _S) {  
        Adjusta(_H, _M, _S);  
    }  
    public void Adjusta(int _H, int _M, int _S) {  
        H = _H; M = _M; S = _S;  
        Normaliza();  
    }  
    public string ParaString() {  
        return string.Format("{0}:{1}:{2}", new object[] {H, M,  
S});  
    }  
    void Normaliza() {  
        M = M + S / 60;  
        S = S % 60;  
        H = H + M / 60;  
        M = M % 60;  
    }  
    public double Hora {  
        get {  
            return H + (M / 60.0) + (S / 3600.0);  
        }  
        set {  
            M = 0; S = 0;  
            H = (int) value;  
            double Sobra = value - H;  
            S = (int) (Sobra * 3600);  
            Normaliza();  
        }  
    }  
    override public string ToString() {  
        return ParaString();  
    }  
}
```

## Interfaz.

Tipo definido por el usuario. Una interfaz es una especie de clase, pero sólo contiene los "prototipos" de los métodos, sin su implementación. Corresponde en C++ a una clase

abstracta con todos los métodos "virtuales puros". Una clase, además de derivarse de otra, puede implementar varias interfaces. Vea un ejemplo:

```
// Declara la interfaz
interfaz IControl
{
    void Paint();
}
// Crea una interfaz derivada
interfaz ITextBox: IControl
{
    void SetText(string text);
}
// La clase implementa la interfaz
class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

## Delegate (Delegar)

Tipo definido por el usuario. Es un "puntero de función orientado a objetos". Podemos atribuir una lista de métodos a un delegate y llamarlos al invocar delegate. Vea un ejemplo:

```
// Declara un delegate. Es un método que no acepta argumentos y retorna
enteros
delegate int MyDelegate();

// Declara una clase. Observe que los métodos tienen la misma "firma" del
delegate de arriba:
// retornan un entero y no aceptan argumentos
public class MyClass
{
    public int InstanceMethod ()
    {
        Console.WriteLine("A message from the instance method.");
        return 0;
    }

    static public int StaticMethod ()
    {
        Console.WriteLine("A message from the static method.");
        return 0;
    }
}

public class MainClass
{
    static public void Main ()
    {
        MyClass p = new MyClass();

        // Transfiere datos del delegate al método de la clase creada
        arriba
```

```

        MyDelegate d = new MyDelegate(p.InstanceMethod);

        // Llama el método vía delegate
        d();

        // Transfiere datos del otro método (ahora es estático)
        d = new MyDelegate(MyClass.StaticMethod);

        // Llama el método vía delegate
        d();
    }
}

```

## Tipo string.

Strings son técnicamente un tipo por referencia, pero poseen algunas características especiales:

- No necesitan ser iniciados con el operador new.
- La atribución de una variable a otra funciona como si copiara el contenido, creando efectivamente otra copia de la variable.
- String que contiene el valor null es un string vacío; no es un error utilizarlo.
- Usted no puede crear una clase derivada de string.

Los strings contienen caracteres Unicode y pueden tener hasta 1G de longitud. Vea un ejemplo:

```

// Declara e inicializa un string
string Name = "Mary";
// Copia para otro string. Si alteramos uno de ellos, el otro mantendrá
su valor
string NewName = Name;
// Atribuye string antiguo
Name = "John";
// Muestra "John - Mary"
System.Console.WriteLine(Name + " - " + NewName);

```

En la segunda parte de este artículo veremos detalles de la implementación de los tipos por referencia y por valor, como uno se convierte en otro ("boxing" y "unboxing") y las ventajas de la unificación del sistema de tipos en el uso de "containers" (contenedores).

## **Tipos de datos en .NET - Parte 2. C#**

Por Mauro Sant'Anna ([mas\\_mauro@hotmail.com](mailto:mas_mauro@hotmail.com)). Mauro es Director regional de MSDN, consultor e instructor de MAS Informática ([www.mas.com.br](http://www.mas.com.br)) y lleva impartiendo formación acerca de la arquitectura .NET desde octubre de 2000.

En el artículo anterior examinamos superficialmente los tipos de datos que hay en .NET. En este artículo veremos algunos detalles sobre cómo se implementan.

### **Tipos por valor.**

Los tipos por valor se almacenan directamente en el disco duro de la computadora. Cuando copiamos un tipo por valor, estamos copiando el valor directamente. No se necesita ninguna memoria además de la estrictamente necesaria para guardar el valor. Estas características cambian los tipos por valor particularmente útiles en tipos pequeños (16 bytes o menos) como enteros, números de punto flotantes y hasta las estructuras pequeñas mismas como una coordenada en el plano (X, Y).

### **Tipos por referencia.**

Los tipos por referencia se almacenan internamente como indicadores. Se deben iniciar necesariamente con el operador nuevo, tienen acceso más lento, gastan algún espacio adicional y añaden presión al administrador de memoria. En retribución, los tipos por referencia pueden contener métodos y otros recursos de OOP como polimorfismo.

Cada categoría de tipos tiene sus ventajas: los que son por valor son más "económicos", al tiempo que los que son por referencia son más flexibles.

La siguiente figura muestra un struct, un tipo por valor y una class (un tipo por referencia). Ambas contienen únicamente dos enteros. Observe cómo la clase que se utiliza es "más costosa" (emplea más recursos):

# Classes e Structs

```
class CPoint { int x, y; ... }  
struct SPoint { int x, y; ... }
```

```
CPoint cp = new CPoint(10, 20);  
SPoint sp = new SPoint(10, 20);
```



## Boxing y Unboxing.

Mientras tanto, en algunas situaciones, sería preferible enfrentar los tipos por valor como referencias, como por ejemplo:

- Llamar métodos para operaciones como conversiones.
- Poner en las clases "containers" (contenedores) como listas y formaciones.
- Tener un tipo capaz de almacenar valores de cualquier otro tipo. Esto es útil en muchas situaciones, como escribir funciones que aceptan diversos tipos de argumentos y en clases "container" como listas y filas.

C# unifica el sistema de tipos a través de un proceso llamado boxing. A través de boxing, puede atribuir un tipo por valor a un tipo object. Cuando hacemos esto, el compilador reserva memoria en forma dinámica para contener el valor y regresa un indicador para este valor. Este indicador puede atribuirse a una variable del tipo object o pasarse como objeto (`this`) para un método.

Boxing proporciona una unificación del sistema de tipos, ya que podemos atribuir cualquier valor a una variable del tipo object.



# Sistema de tipos unificado

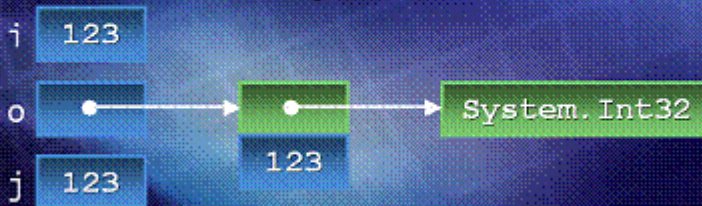
## ◆ Boxing

- ❖ Aloca e copia valor

## ◆ Unboxing

- ❖ Verifica tipo e copia valor

```
int i = 123;  
object o = i;  
int j = (int)o;
```



Podemos recuperar el valor colocado en la variable del tipo object a través de un cast, como vemos en la figura anterior.

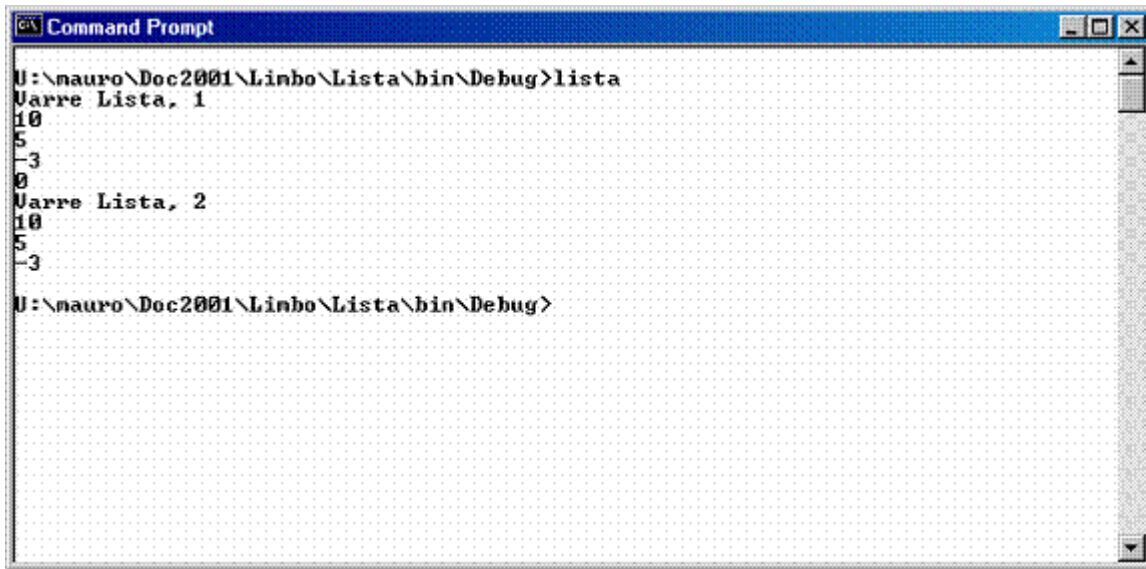
## Contenedores.

Una de las grandes ventajas de la unificación del sistema de tipos, es la posibilidad de definir clases de "containers" que pueden almacenar valores de cualquier tipo. Una de esas clases es `ArrayList`, presente en namespace `System.Collections`. El siguiente ejemplo muestra diversas operaciones en `ArrayList`:

```
using System.Collections;  
...  
// Crea la clase contenedor  
ArrayList Lista = new ArrayList();  
// Agrega algunos enteros  
Lista.Add(10); Lista.Add(5); Lista.Add(-3); Lista.Add(0);  
Console.WriteLine("Varre Lista, 1");  
// Examina toda la lista (versión 1)  
foreach(int Valor in Lista) {  
    // Muestra el Valor  
    Console.WriteLine(Valor);  
}  
Console.WriteLine("Varre Lista, 2");
```

```
// Examina toda la lista (versión 2)
for(int i = 0; i < Lista.Count - 1; i++) {
    // Incluye el valor
    int x = (int) Lista[i];
    // Muestra el Valor
    Console.WriteLine(x);
}
```

Este es el resultado del programa anterior:



```
U:\mauro\Doc2001\Linbo\Lista\bin\Debug>lista
Varre Lista, 1
10
5
-3
Varre Lista, 2
10
5
-3
U:\mauro\Doc2001\Linbo\Lista\bin\Debug>
```

El ejemplo anterior contiene únicamente números enteros, pero ArrayList puede contener valores de cualquier tipo. Vea el ejemplo siguiente, que almacena valores de diversos tipos. Observe la verificación del tipo antes de sacar el valor de la formación:

```
using System.Collections;
...
public class Class2 {
    public static int Main(string[] args) {
        // Crea la clase contenedor
        ArrayList Lista = new ArrayList();
        // Agrega algunos valores diversos
        Lista.Add(10); Lista.Add(5.7); Lista.Add(-3.9m); Lista.Add("casa");
        // Examina toda la lista (versión 1)
        Console.WriteLine("Varre Lista, 1");
        foreach(object Valor in Lista) {
            // Muestra el Valor
            Console.WriteLine(Valor);
        }
        Console.WriteLine("Varre Lista, 2");
        // Examina toda la lista (versión 2)
        for(int i = 0; i < Lista.Count - 1; i++) {
            // Incluye el valor
            object x = Lista[i];
            // Incluye sólo si es un entero
            if (x.GetType() == typeof(int)) {
```

```

        // Hace cast para incluir el entero
        int Valor = (int) x;
        Console.WriteLine(Valor);
    }
}
}
}

```

Este es el resultado del programa anterior:

```

U:\mauro\Doc2001\Linbo\Lista2\bin\Debug>lista2
Varre Lista, 1
10
5.7
-3.9
casa
Varre Lista, 2
10
U:\mauro\Doc2001\Linbo\Lista2\bin\Debug>_

```

## Conclusión.

La unificación del sistema de tipos es una de las novedades realmente únicas de .NET. Esa unificación resuelve, en forma brillante, algunos de los problemas encontrados en el desarrollo de software.