

INTRODUCCION A LA PROGRAMACION C#

2 - MÓDULO 2 : ARRAYS, STRINGS Y CONSTANTES

UNIDAD: 2

MODULO: 2

PRESENTACIÓN: En esta unidad se explicarán los conceptos de arrays, strings y constantes en el lenguaje C#.

OBJETIVOS

Que los participantes logren: Comprender los conceptos de esta unidad y aplicar en la practica dichos conceptos.

TEMARIO

La clase System.Array.....	4
Strings (o Cadenas de texto).....	6
Comparación de strings	6
Comparación de strings por referencia.....	7
Caso particular de la comparación de strings por referencia.....	8
Concatenación de strings.....	9
Trabajando con strings modificables.....	11
Otras utilidades de System. String	11
Observaciones sobre las operaciones con strings	13
Constantes	14

La clase `System.Array`

En realidad, todas las tablas que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de `System.Array`.

Es decir, van a disponer de todos los miembros que se han definido para esta clase, entre los que son destacables:

- **Length** : Campo de sólo lectura que informa del número total de elementos que contiene la tabla.

Si la tabla tiene más de una dimensión o nivel de anidación indica el número de elementos de todas sus dimensiones y niveles.

Por ejemplo:

```
int[] tabla = { 1, 2, 3, 4 };
int[][] tabla2 = { new int[] { 1, 2 }, new int[] { 3, 4, 5 } };
int[,] tabla3 = { { 1, 2 }, { 3, 4 } };
Console.WriteLine(tabla.Length); //Imprime 4
Console.WriteLine(tabla2.Length); //Imprime 2
Console.WriteLine(tabla3.Length); //Imprime 4
```

- **Rank** : Campo de sólo lectura que almacena el número de dimensiones de la tabla.

Por ejemplo:

```
int[] tabla = { 1, 2, 3, 4 };
int[][] tabla2 = { new int[] { 1, 2 }, new int[] { 3, 4, 5 } };
int[,] tabla3 = { { 1, 2 }, { 3, 4 } };
Console.WriteLine(tabla.Rank); //Imprime 1
Console.WriteLine(tabla2.Rank); //Imprime 1
Console.WriteLine(tabla3.Rank); //Imprime 2
```

- **GetLength(int dimensión)** : Método que devuelve el número de elementos de la dimensión especificada.

Las dimensiones se indican empezando a contar desde cero, por lo que si quiere obtenerse el número de elementos de la primera dimensión habrá que usar `GetLength(0)` , si se quiere obtener los de la segunda habrá que usar `GetLength(1)` , etc.

Por ejemplo:

```
int[,] tabla = {{1,2}, {3,4,5,6}};  
Console.WriteLine(tabla.GetLength(0)); // Imprime 2  
Console.WriteLine(<gt;tabla.GetLength(1)); // Imprime 4
```

- **CopyTo(Array destino, int posición)** : Copia todos los elementos de la tabla sobre la que es aplica en la que se le pasa como primer parámetro a partir de la posición de la misma indicada como segundo parámetro.

Por ejemplo:

```
int[] tabla1 = {1,2,3,4};  
int[] tabla2 = {5,6,7,8};  
tabla1.CopyTo(tabla2,0); // A partir de ahora, ambas tablas contienen  
{1,2,3,4}
```

Ambas tablas han de ser unidimensionales. Por otro lado, y como es obvio, la tabla de destino ha de ser de un tipo que pueda almacenar los objetos de la tabla fuente, el índice especificado ha de ser válido (mayor o igual que cero y menor que el tamaño de la tabla de destino) y no ha de valer *null* ninguna. Si no fuese así, saltarían excepciones de diversos tipos informando del error.

Aparte de los miembros aquí señalados, de *System.Array* cuenta con muchos más que permiten realizar tareas tan frecuentes como búsquedas de elementos, ordenaciones, etc.

Strings (o Cadenas de texto)

Una cadena de texto (o string como la llamamos los programadores) no es más que una secuencia de caracteres Unicode. En C# se representan mediante objetos del tipo de dato llamado *string*, que no es más que un alias del tipo `System.String`.

Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas. Ejemplos de ambos casos se muestran a continuación:

```
string cadena1 = "John Wayne";  
string cadena2 = cadena1;
```

En el primer caso se ha creado un objeto *string* que representa a la cadena formada por la secuencia de caracteres John Wayne indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan sino que sólo se usan como delimitadores de la misma).

En el segundo caso la variable `cadena2` creada se genera a partir de la variable `cadena1` ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Comparación de strings

Hay que tener en cuenta que el tipo *string* es un tipo referencia, por lo que en principio la comparación entre objetos de este tipo debería comparar sus direcciones de memoria como pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos que esto no ocurre en el caso de las cadenas:

```
using System;  
public class IgualdadCadenas  
{  
    public static void Main()  
    {  
        string cadena1 = "John Wayne";  
        string cadena2 = String.Copy(cadena1);  
        Console.WriteLine(cadena1 == cadena2);  
    }  
}
```

El método *Copy()* de la clase *String* usado devuelve una copia del objeto que se le pasa como parámetro. Por tanto, al ser objetos diferentes se almacenarán en posiciones distintas de memoria y al compararlos debería devolverse *false* como pasa con cualquier tipo referencia.

Sin embargo, si ejecuta el programa verá que lo que se obtiene es precisamente lo contrario: *true*. Esto se debe a que para hacer para hacer más intuitivo el trabajo con cadenas, en C# se ha modificado el operador de igualdad para que cuando se aplique entre cadenas se considere que sus operandos son iguales sólo si son lexicográficamente equivalentes y no si referencian al mismo objeto en memoria.

Además, esta comparación se hace teniendo en cuenta la capitalización usada, por lo que

```
Console.WriteLine("hola" == "Hola");  
Console.WriteLine("HOLA" == "Hola");
```

devolverán *False* ya que contienen las mismas letras pero con distinta capitalización.

Comparación de strings por referencia

Si se quisiese comparar cadenas por referencia habría que optar por una de estas dos opciones:

1. compararlas con *Object.ReferenceEquals()*
2. convertirlas en objects y luego compararlas con *==* Por ejemplo:

```
Console.WriteLine(Object.ReferenceEquals(cadena1, cadena2));  
Console.WriteLine((object)cadena1 == (object)cadena2);
```

Ahora sí que lo que se comparan son las direcciones de los objetos que representan a las cadenas en memoria, por lo que la salida que se mostrará por pantalla es:

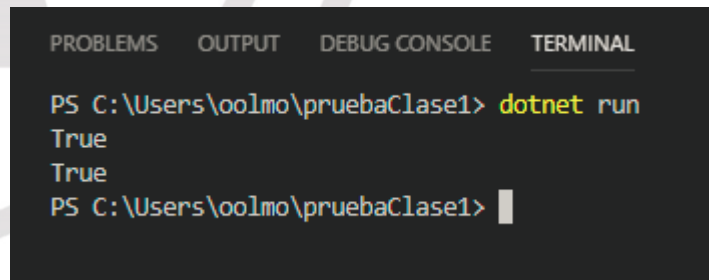
```
PS C:\Users\oolmo\pruebaClase1> dotnet run  
False  
False
```

Caso particular de la comparación de strings por referencia

Hay que señalar una cosa, y es que aunque en principio el siguiente código debería mostrar la misma salida por pantalla que el anterior ya que las cadenas comparadas se deberían corresponder a objetos que aunque sean lexicográficamente equivalentes se almacenan en posiciones diferentes en memoria:

```
using System;
public class IgualdadCadenas2
{
    public static void Main()
    {
        string cadena3 = "Harrison Ford";
        string cadena4 = "Harrison Ford";
        Console.WriteLine(Object.ReferenceEquals(cadena3, cadena4));
        Console.WriteLine(((object)cadena3) == ((object)cadena4));
    }
}
```

Si lo ejecutamos veremos que la salida obtenida es justamente la contraria:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\oolmo\pruebaClase1> dotnet run
True
True
PS C:\Users\oolmo\pruebaClase1> |
```

Esto se debe a que **el compilador** ha detectado que ambos literales de cadena son lexicográficamente equivalentes y ha decidido que **para ahorrar memoria** lo mejor es almacenar en memoria **una única copia** de la cadena que representan y hacer que **ambas variables apunten a esa copia común**.

Concatenación de strings

Al igual que el significado del operador `==` ha sido especialmente modificado para trabajar con cadenas, lo mismo ocurre con el operador binario `+`. En este caso, cuando se aplica entre dos cadenas o una cadena y un carácter lo que hace es devolver una nueva cadena con el resultado de concatenar sus operandos.

Así por ejemplo, en el siguiente código las dos variables creadas almacenarán la cadena *Hola Mundo* :

```
public class Concatenacion
{
    public static void Main()
    {
        string cadena = "Hola" + " Mundo";
        string cadena2 = "Hola Mund" + 'o';

        System.Console.WriteLine(cadena);
        System.Console.WriteLine(cadena2);
    }
}
```

Por otro lado, el acceso a las cadenas se hace de manera similar a como si de tablas de caracteres se tratase: su “campo” *Length* almacenará el número de caracteres que la forman y para acceder a sus elementos se utiliza el operador `[]`. Por ejemplo, el siguiente código muestra por pantalla cada carácter de la cadena *Hola* en una línea diferente:

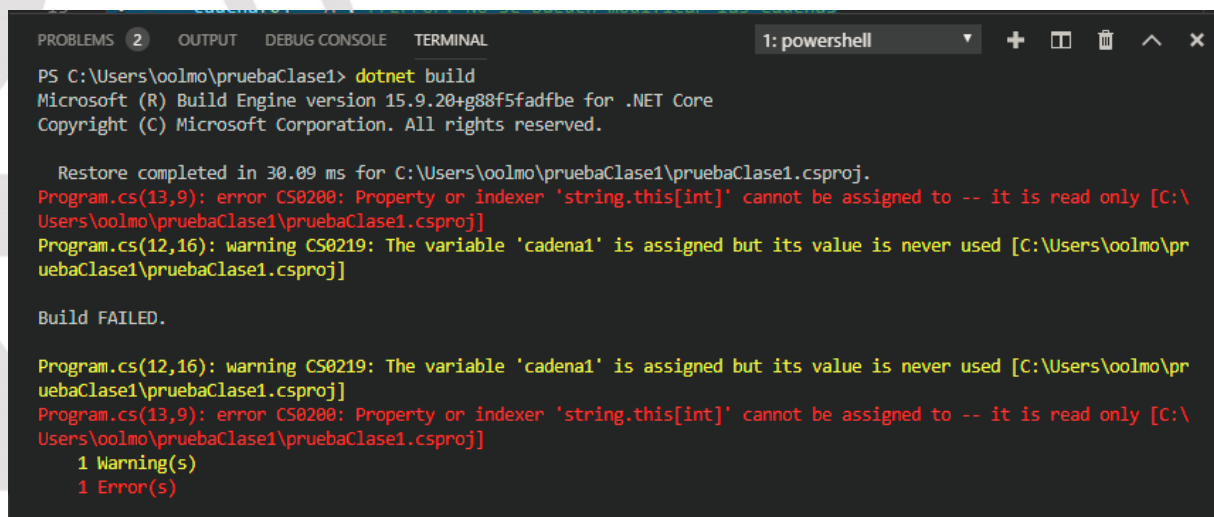
```
using System;
public class AccesoCadenas
{
    public static void Main()
    {
        string cadena = "Hola";
        Console.WriteLine(cadena[0]);
        Console.WriteLine(cadena[1]);
        Console.WriteLine(cadena[2]);
        Console.WriteLine(cadena[3]);
    }
}
```

Sin embargo, hay que señalar una diferencia importante respecto a la forma en que se accede a las tablas: **las cadenas son inmutables**, lo que significa que no es posible modificar los caracteres que las forman.

Esto se debe a que el compilador comparte en memoria las referencias a literales de cadena lexicográficamente equivalentes para así ahorrar memoria, y si se permitiese modificarlos los cambios que se hiciesen a través de una variable a una cadena compartida afectarían al resto de variables que la compartan, lo que podría causar errores difíciles de detectar.

Por tanto, hacer esto es incorrecto:

```
string cadena1 = "Hola";  
cadena[0] = "A"; //Error: No se pueden modificar las cadenas
```



The screenshot shows a terminal window with the following output:

```
PS C:\Users\oolmo\pruebaClase1> dotnet build  
Microsoft (R) Build Engine version 15.9.20+g88f5fadbfe for .NET Core  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Restore completed in 30.09 ms for C:\Users\oolmo\pruebaClase1\pruebaClase1.csproj.  
Program.cs(13,9): error CS0200: Property or indexer 'string.this[int]' cannot be assigned to -- it is read only [C:\Users\oolmo\pruebaClase1\pruebaClase1.csproj]  
Program.cs(12,16): warning CS0219: The variable 'cadena1' is assigned but its value is never used [C:\Users\oolmo\pruebaClase1\pruebaClase1.csproj]  
  
Build FAILED.  
  
Program.cs(12,16): warning CS0219: The variable 'cadena1' is assigned but its value is never used [C:\Users\oolmo\pruebaClase1\pruebaClase1.csproj]  
Program.cs(13,9): error CS0200: Property or indexer 'string.this[int]' cannot be assigned to -- it is read only [C:\Users\oolmo\pruebaClase1\pruebaClase1.csproj]  
1 Warning(s)  
1 Error(s)
```

Sin embargo, el hecho de que no se puedan modificar las cadenas no significa que no se puedan cambiar los objetos almacenados en las variables de tipo *string*. Por ejemplo, el siguiente código es válido:

```
string cadena = "Hola";  
cadena = "Adios";
```

Trabajando con strings modificables

Si se desea trabajar con cadenas modificables puede usarse *System.Text.StringBuilder*, que funciona de manera similar a *string* pero permite la modificación de sus cadenas en tanto que estas no se comparten en memoria.

Para crear objetos de este tipo basta pasar como parámetro de su constructor el objeto *string* que contiene la cadena a representar mediante un *StringBuilder*, y para convertir un *StringBuilder* en *string* siempre puede usarse su método *ToString()* heredado de *System.Object*.

Por ejemplo:

```
using System.Text;
using System;
public class ModificaciónCadenas
{
    public static void Main()
    {
        StringBuilder cadenaMutable = new StringBuilder("Fideos");
        String cadenaInmutable = cadenaMutable.ToString();
        cadenaMutable[0] = 'V';
        Console.WriteLine(cadenaMutable); // Muestra Videos
        Console.WriteLine(cadenaInmutable); // Muestra Fideos
    }
}
```

Otras utilidades de System. String

Aparte de los métodos ya vistos, en la clase *System.String* se definen muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

int IndexOf(string substring):

Indica cuál es el índice de la primera aparición de la substring indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha substring se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo *int* cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo *int* es posible indicar algún índice anterior donde terminarla.

Nótese que es un método muy útil para saber si una cadena contiene o no alguna substring determinada, pues sólo si no la encuentra devuelve un -1 .

int LastIndexOf(string substring) :

Funciona de forma similar a *IndexOf()* sólo que devuelve la posición de la última aparición de la substring buscada en lugar de devolver la de la primera.

string Insert(int posición, string substring) :

Devuelve la cadena resultante de insertar la substring indicada en la posición especificada de la cadena sobre la que se aplica.

string Remove(int posición, int número) :

Devuelve la cadena resultante de eliminar el número de caracteres indicado que hubiese en la cadena sobre al que se aplica a partir de la posición especificada.

string Replace(string aSustituir, string sustituta) :

Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparición de la cadena a sustituir indicada por la cadena sustituta especificada como segundo parámetro.

string Substring(int posición, int número) :

Devuelve la substring de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la substring que va desde la posición indicada hasta el final de la cadena.

string ToUpper() y string ToLower() :

Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

Observaciones sobre las operaciones con strings

Es preciso incidir en que aunque hayan métodos de inserción, reemplazo o eliminación de caracteres que puedan dar la sensación de que es posible modificar el contenido de una cadena, en realidad las cadenas son inmutables y dicho métodos lo que hacen es devolver una nueva cadena con el contenido correspondiente a haber efectuado las operaciones de modificación solicitadas sobre la cadena a la que se aplican.

Por ello, las cadenas sobre las que se aplican quedan intactas como muestra el siguiente ejemplo:

```
using System;
public class EjemploInmutabilidad
{
    public static void Main()
    {
        string cadena1 = "Hola";
        string cadena2 = cadena1.Remove(0, 1);
        Console.WriteLine(cadena1);
        Console.WriteLine(cadena2);
    }
}
```

La salida por pantalla de este ejemplo demuestra lo antes dicho, pues es:

```
PS C:\Users\oolmo\pruebaClase1> dotnet run
Hola
ola
PS C:\Users\oolmo\pruebaClase1> 
```

Como se ve, tras el *Remove()* la cadena1 permanece intacta y el contenido de cadena2 es el que debería tener cadena1 si se le hubiese eliminado su primer carácter.

Constantes

Una constante es una variable cuyo valor puede determinar el compilador durante la compilación y puede aplicar optimizaciones derivadas de ello. Para que esto sea posible se ha de cumplir que el valor de una constante no pueda cambiar durante la ejecución, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante. Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador *const* y dándoles siempre un valor inicial al declararlas. O sea, con esta sintaxis:

```
const <tipoConstante> <nombreConstante> = <valor>;
```

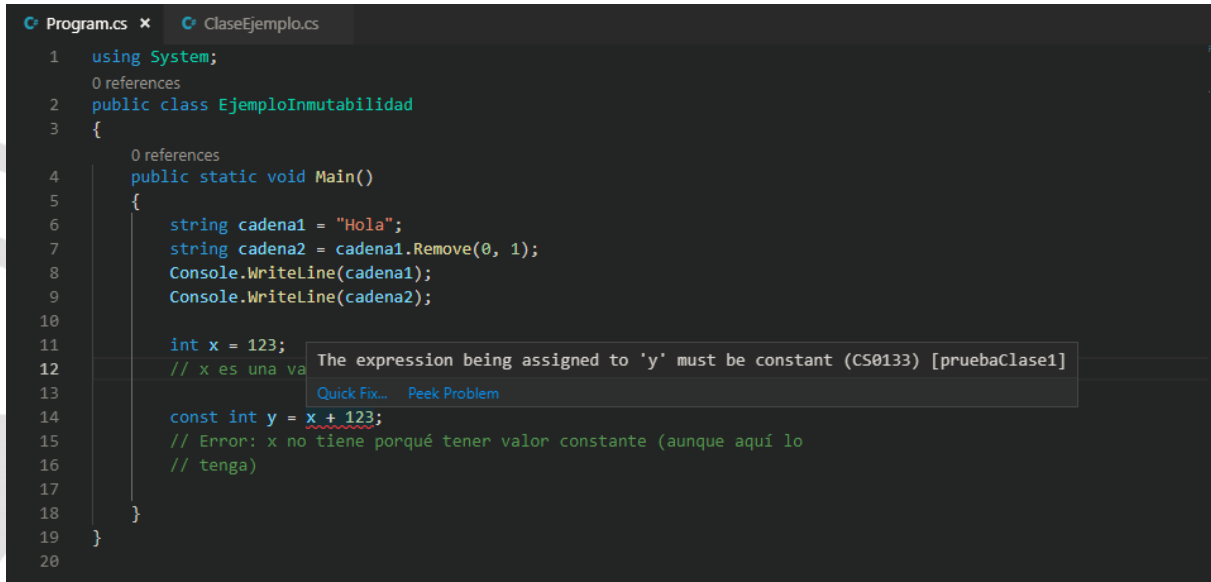
Así, ejemplos de definición de constantes es el siguiente:

```
const int a = 123;  
const int b = a + 125;
```

Dadas estas definiciones de constantes, lo que hará el compilador será sustituir en el código generado todas las referencias a las constantes a y b por los valores 123 y 248 respectivamente, por lo que el código generado será más eficiente ya que no incluirá el acceso y cálculo de los valores de a y b .

Nótese que puede hacer esto porque en el código se indica explícitamente cual es el valor que siempre tendrá a y, al ser este un valor fijo, puede deducir cuál será el valor que siempre tendrá b . Para que el compilador pueda hacer estos cálculos se ha de cumplir que el valor que se asigne a las constantes en su declaración sea una expresión constante.

Por ejemplo, el siguiente código no es válido en tanto que el valor de x no es constante:



```
1 using System;
2 public class EjemploInmutabilidad
3 {
4     public static void Main()
5     {
6         string cadena1 = "Hola";
7         string cadena2 = cadena1.Remove(0, 1);
8         Console.WriteLine(cadena1);
9         Console.WriteLine(cadena2);
10
11         int x = 123;
12         // x es una variable
13
14         const int y = x + 123;
15         // Error: x no tiene porqué tener valor constante (aunque aquí lo
16         // tenga)
17     }
18 }
19
20
```

Debido a la necesidad de que el valor dado a una constante sea precisamente constante, no tiene mucho sentido crear constantes de tipos de datos no básicos, pues a no ser que valgan *null* sus valores no se pueden determinar durante la compilación sino únicamente tras la ejecución de su constructor.

La única excepción a esta regla son los tipos enumerados, cuyos valores se pueden determinar al compilar como se explicará cuando los veamos el tema Enumeraciones

Todas las constantes son implícitamente estáticas, por lo se considera erróneo incluir el modificador *static* en su definición al no tener sentido hacerlo.

BIBLIOGRAFÍA RECOMENDADA:

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/types/index>

Head First C# 3rd Edition Jennifer Greene Andrew Stellman

Fecha	Versión	Observaciones
09/07/2018	1.0	Versión Original
11/07/2018	1.1	Se corrige typo en pagina 23