Los Tipos de C#.

Introducción

La mayoría de los lenguajes orientados a objetos tienen dos tipos distintos de datos: aquellos que son intrínsecos al lenguaje (tipos primitivos) y aquellos que pueden ser creados por el programador. Los tipos primitivos son tipos simples como caracteres, strings y números. Es evidente que tener dos tipos de datos o variables es una fuente de problemas ya que dificulta el tratamiento de estructuras con datos genéricos, como cuando, por ejemplo, se quiere especificar que un determinado método pueda recibir un argumento de cualquier tipo. Como los tipos primitivos son incompatibles entre sí, no se puede trabajar con un método de estas características a menos que se escriba una clase que envuelva a cada uno de los tipos primitivos, como ocurre en Java.

Sin embargo, en C# todo objeto deriva implícitamente de una simple clase base, que se llama `System.Object`. Esto quiere decir que **todas las variables en C# son, directa o indirectamente, objetos**. Se hablará más adelante de esta clase base.

Sin embargo, no siempre es eficaz trabajar en programación con objetos y, por esta razón, los creadores de la plataforma Microsoft.NET separaron los tipos en dos categorías: ***tipos valor*** y ***tipos referencia***.

Los tipos valor son:

a) Los tipos simples – como `char`, `int`, `float`, `etc...`- predefinidos por el sistema.
b) Las enumeraciones: tipos `enum`,
c) Las estructuras: tipos `struct`.

Tanto los tipos `enum` como `struct` pueden ser predefinidos por el sistema o por el programador.

Los tipos referencia son:

a) Las clases –el tipo `class`- entre las que se incluyen los tipos `string` y `object`.
b) Las interfaces –el tipo `interface-`.
c) El tipo `delegate`,
d) Los arrays -el tipo `array`-

Además, existe una tercera categoría que son los **punteros**, que están disponibles únicamente para "código inseguro". Los punteros en C# y, en general, el código inseguro se tratan con profundidad en un capítulo posterior.

Las diferencias más importantes entre los *tipos valor* y los *tipos referencia* se pueden resumir así:

- Las variables de *tipo valor* contienen directamente valores o **datos** –como una variable de C o C++-, mientras que las variables de *tipo referencia* almacenan **referencias** a datos, que en realidad son objetos.
- Las variables de *tipo valor* se almacenan directamente en la *pila* o *stack* mientras que las variables de *tipo referencia* son referencias que se

almacenan en la *pila* y apuntan o referencian a un objeto que se almacena en el *heap* o *montón* y que contiene toda la información referente a ese objeto.

- Todo tipo en C# deriva directa o indirectamente del tipo clase `System.Object`, que es la clase raíz de la jerarquía de herencia en C#. Esto quiere decir que una variable tanto de tipo valor como de tipo referencia puede ser tratada como un objeto.
- En los *tipo valor* cada variable tiene su propia copia de los datos y las operaciones sobre una variable no afectan a otras variables. En cambio, en los de *tipo referencia* puede darse el caso de que dos variables de *tipo referencia* apunten o referencien al mismo objeto, con lo que las operaciones realizadas a través de una de las referencias pueden llegar a afectar al objeto referenciado por la otra.

Los desarrolladores pueden definir nuevos tipos valor por medio de declaraciones de estructuras –`struct`- y enumeraciones –`enum`- y nuevos tipos referencia por medio de clases –`class`-, interfaces –`interface`- y delegates –`delegate`-.

Características del lenguaje

Comentarios

Los comentarios se utilizan para que una o varias líneas sean ignoradas por el compilador. Estas líneas se utilizan, en general, para hacer que el código sea más comprensible.

Hay dos tipos de comentarios:

a) El símbolo `//` convierte en comentario todo lo que esté situado a su derecha y en la misma línea.
b) Todo el texto encerrado entre los símbolos `/*` y `*/` será considerado como comentario por el programador.

Por ejemplo:

```
using System;
namespace ConsoleApplication3
{
      /*****************COMENTARIO*************
       * Estas tres líneas son ignoradas por el compilador
       *****************CODIGO*****************/

      class MiClase
      {
            static void Main(string[] args)
            {
                  //La siguiente línea se imprimirá en pantalla
                  Console.WriteLine("Esto se imprimirá");
            }

      }
      //Este es otro comentario
}
```

Identificadores

Un identificador es un nombre que se utiliza para nombrar, etiquetar o "identificar" una variable, una clase, un método, etc.

C# es sensible a las mayúsculas. Esto quiere decir que distinguirá, por ejemplo, entre `PI`, `pi` o `Pi`.

Los identificadores deben comenzar por una letra o por guión bajo y se puede utilizar cualquier carácter UNICODE o número en dichos identificadores. Por eso, por fin pueden utilizarse como nombre de variable palabras con ñ o con acento. Así, las siguientes palabras son identificadores válidos:

```
unaVariable
OtraVariable
variable1
variable_2
unNiño
Niño2
Rectángulo
```

Sin embargo no pueden utilizarse como identificadores las palabras siguientes, reservadas por el lenguaje:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	lock	is	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Tabla 3.1

Literales

Los literales son los valores que pueden tener o almacenar los tipos básicos del lenguaje. Se tratarán con profundidad al hablar de los tipos de datos. Pueden ser:

a) enteros
b) reales
c) booleanos o lógicos
d) caracteres
e) cadenas

f) `null`. Este literal indica que una referencia no apunta a ningún objeto.

Tipos valor .

Los datos de tipo valor se dividen en tres categorías principales:

- Tipo simple
- Tipo estructura
- Tipo enumeración

Los tipos simples pueden clasificarse del siguiente modo.

- Tipo Entero
  - con signo
    - `sbyte`
    - `short`
    - `int`
    - `long`
  - sin signo
    - `byte`
    - `ushort`
    - `uint`
    - `ulong`
- Tipo coma flotante
  - `float`
  - `double`
- Tipo `char`
- Tipo booleano: `bool`
- Tipo `decimal`

Una variable de *tipo valor* contiene y almacena un dato del tipo que se especifique: `int`, `float`, `double`, etc... Se habla entonces de variable de tipo entero, `float`, `double`... Una variable *de tipo valor* no puede ser `null`.

Por ejemplo:

```
int unEntero=4;         //variable de tipo entero
double unDouble=2.3;    // variable de tipo double
```

Todos los *tipo valor* heredan implícitamente de la clase `System.Object` -que es la clase raíz de la jerarquía de herencia en C#-. Por eso, todas las variables en C# son objetos y tienen los miembros que heredan de `object`. Por ejemplo, en C# es correcto escribir:

```
Console.WriteLine(7.ToString());    //7 es un objeto que utiliza
                                    //el método ToString()
                                    //heredado de object


int i = int.MaxValue;         //System.Int32.MaxValue,
```

```
int n = Int32.MinValue;
string s = i.ToString();      //System.Int32.ToString()
string t = 123.ToString();    //System.Int32.ToString()
```

Sin embargo, no es posible crear tipos derivados de un *tipo valor* aunque, como en los *tipo referencia*, pueden implementar interfaces. Más adelante, en un capítulo posterior se estudiarán estos conceptos con más detenimiento.

Declaración de variables

Cuando se *declara* una variable de *tipo valor* se reserva memoria para el número de bytes asociado con ese tipo de dato *en la pila* y se trabaja directamente con ese conjunto de bits. A continuación, se *asocia* a ese espacio en memoria el nombre de la variable de tipo valor y se *almacena* el valor o valores que se asigna a la variable en ese espacio de memoria. Por ejemplo, si se escribe:

```
int unValor = 78;
```

En la expresión anterior se reserva espacio en memoria para 32 bits en la pila o stack, ya que cada variables de tipo entero –como se puede ver en la tabla 3.4- ocupa 4 bytes y ese espacio se "etiqueta" o asocia con una variable denominada `unValor`. Además, el valor `78` se almacena en esos 32 bits, es decir, se hace una copia de ese valor en esas posiciones de memoria.

A continuación se adjuntan ejemplos de inicializaciones:

```
int unEntero = 7;

double unDouble = 3.141592;

char unChar = 'g';

bool unBoolean = true;
```

Inicialización por defecto de los tipos valor.

Cada variable de *tipo valor* tiene un constructor implícito por defecto que la inicializa en caso de que no lo haga el usuario. Todos los tipos valor tienen un constructor público sin parámetros llamado **constructor por defecto**. El constructor por defecto devuelve una instancia inicializada a la que se conoce como *valor por defecto*. La tabla 3.2 representa dichos valores de inicialización:

Tipo valor	**Valor por defecto**
`bool`	`false`
`byte`	`0`
`char`	`'\0'`
`decimal`	`0.0M`
`double`	`0.0D`
`enum`	`0`
`float`	`0.0F`

```
int                 0
long                0L
sbyte               0
short               0
uint                0
ulong               0
ushort              0
```

Tabla 3.2

En el caso de las estructuras –`struct`-, el valor por defecto es el resultado de dar el valor por defecto a cada uno de sus campos y a `null` todos los tipos referencia.

Alias de los tipos valor

Para simplificar el lenguaje, .NET permite invocar por medio de un "alias" a algunos tipos de datos. Por ejemplo, en lugar de decir que una variable es de tipo `System.Int32`**,** se le invoca por su alias, en este caso `int`.

Por eso, las dos sentencias que siguen son equivalentes:

```
System.Int32 unEntero = 34;

int unEntero = 34
```

La lista completa de "alias" se presenta en la tabla 3.3.

Tipo de .NET	**Tipo de C# o alias**
`System.Boolean`	`bool`
`System.Byte`	`byte`
`System.Sbyte`	`sbyte`
`System.Char`	`char`
`System.Decimal`	`decimal`
`System.Double`	`double`
`System.Single`	`float`
`System.Int32`	`int`
`System.UInt32`	`uint`
`System.Int64`	`long`
`System.UInt64`	`ulong`
`System.Object`	`object`
`System.Int16`	`short`
`System.UInt16`	`ushort`
`System.String`	`string`

Tabla 3.3

Clasificación de tipos Valor

Los tipos valor se dividen en tipos simples, estructuras y enumeraciones. A continuación se estudian brevemente cada uno de los tipos simples y posteriormente se hablará de las enumeraciones y de las estructuras.

Tipos Enteros

C# soporta ocho tipos de enteros predefinidos. En la tabla 3.4 se especifican cada uno de ellos, la cantidad de memoria que se reserva en la pila y el rango de valores que pueden tomar.

Tipo	**descripción**	**Bytes**	**Rango**
`sbyte`	Tipo entero con signo	1	`de -128 a 127`
`byte`	Tipo entero sin signo	1	`de 0 a 255`
`short`	Tipo entero con signo	2	`de -32,768 a 32,767`
`ushort6`	Tipo entero sin signo	2	`de 0 a 65,535`
`int`	Tipo entero con signo	4	`de -2,147,483,648 a 2,147,483,647`
`uint`	Tipo entero sin signo	4	`de 0 a 4,294,967,295`
`long`	Tipo entero con signo	8	`de –9,223,372,036,854,775,808 a 9,223,372,036,854,775,807`
`ulong`	Tipo entero sin signo	8	`de 0 a 18,446,744,073,709,551,615`

Tabla 3.4

Los números enteros son muy importantes pero también quizá los más conocidos en cualquier lenguaje de programación. A todas las variables de tipo integral se les puede asignar valores en notación decimal o hexadecimal. Estos últimos requieren del prefijo `0x`.

Si hay ambigüedad acerca de si un determinado tipo entero es `int`, `uint`, `long` o `ulong`, por defecto se convierte a `int`.

Para especificar el tipo de dato entero que se está tratando se debe añadir al final del valor `U` o `L`.

Por ejemplo:

```
uint unaVariableSinSigno=652U;
long unLong=836L;
ulong unUlong=6546UL;
```

Tipos en coma flotante

C# soporta números en coma flotante, como `float` o `double`. Difieren entre ellos en el rango de valores que pueden tomar y en la precisión.

En la tabla 3.5 se especifican estos valores y el espacio de memoria que requieren los tipos en coma flotante. `float` tiene menor precisión que `double`. Tiene 7 dígitos de precisión y `double` 14 o 15 dígitos.

Como regla general, hay que tener en cuenta que si en una determinada expresión hay tipos coma flotante y enteros, éstos últimos se convierten a coma flotante antes de llevar a cabo el cálculo.

Si hubiese ambigüedad acerca de si un determinado valor en coma flotante es `float` o `double`, por defecto se convierte a `double`. Si se quiere especificar que un determinado valor es `float`, se debe añadir el carácter `F` (ó `f`) como por ejemplo:

```
float unFloat=29.23F;
```

Es posible trabajar con C# en aplicaciones científicas, porque permite gran variedad de posibilidades y precisión en los cálculos.

`float`	Tipo real. 7 díg. de precisión	4	de $\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$
`double`	Tipo real. 15 díg. de precisión	8	de $\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$

Tabla 3.5

Tipo decimal

Hay ocasiones en las que es preciso trabajar con números con muchas cifras sin perder precisión en los cálculos. Esto sucede en las aplicaciones financieras que necesitan trabajar con números fraccionarios muy grandes. A veces, los tipos `float` o `double` pueden no ser útiles para este tipo de programas. Para ello, C# proporciona el tipo `decimal`, que es un tipo de dato de gran precisión, de 128 bits.

En la tabla 3.6 se recoge una comparación de los tipos en coma flotante y decimal y se especifica el número de *bytes* que ocupan en memoria, el número de dígitos de precisión y el rango de valores que ofrece.

Tipo	***Bytes***	**Precisión**	**Rango**
float	4	7 dígitos	de $1,5\times10^{-45}$ a $3,4\times10^{38}$
double	8	15-16 dígitos	de $5,0\times10^{-324}$ a $1,7\times10^{308}$
decimal	16	28-29 dígitos	de $1,0\times10^{-28}$ a $7,9\times10^{28}$

Tabla 3.6**.** Tipos de datos que permiten almacenar valores en coma flotante.

Aparentemente el rango de valores en coma flotante es inmenso, pero se debe tener cuidado pues la forma en que se representan impide que los cálculos con números grandes tengan la precisión que en muchos casos se puede necesitar. Por ejemplo, el tipo `float` aparentemente almacena números en el rango $1,5\times10^{-45}$ a $3,4\times10^{38}$, pero

en absoluto permite representar todos los números dentro de este rango. La explicación es simple: para representar estos números se utilizan cuatro bytes y esta cantidad de almacenamiento permite representar `4.294.967.295` valores diferentes, ya sean enteros o reales. Por ejemplo, el valor `0.98765432109876543210`, aunque se encuentra dentro del rango de valores del tipo `double` no se puede representar con precisión y C# lo redondea. El tipo decimal permite una enorme precisión, pero es importante especificar que esta precisión se ofrece en dígitos, no en lugares decimales. Si se necesita gran precisión para los cálculos, entonces lo más probable es que haya que utilizar el tipo de datos `decimal`, aunque también tiene que tener en cuenta que las operaciones con los valores de este tipo son mucho más lentas que las operaciones con los valores de otros tipos. El intervalo de tipo `decimal` es mucho menor que la del tipo `double` pero la precisión es mucho mayor. Es fácil imaginar los problemas que podría generar los grandes números de una gran entidad bancaria.

Para especificar que se trata de un tipo decimal, se debe especificar con la letra `M` (ó `m`) al final del número.

Por ejemplo:

```
float unFloat=7.0F;
double unDouble=5.89;
decimal unDecimal=6.56M;
```

Tipo booleano

Admite dos posibles valores:

- `True` (a diferencia de C y C++ no es un valor distinto de cero).
- `False`

Si una variable se declara como `bool`, sólo puede tomar valor `True` o `False`. No se puede convertir un valor booleano a un valor entero.

Por eso, en C# no es correcto el siguiente código:

```
int a=0;
if(a)             //ERROR. Debe escribirse if(a==0)
      Console.WriteLine("a no es cero");
else
      Console.WriteLine("a es cero");
```

El tipo `bool` ocupa 4 bytes.

Tipo char

C# soporta también el tipo `char`. Representa un carácter UNICODE, con una longitud de 16 bits, o sea, ocupa 2 bytes de memoria. De esta forma, en C# puede representarse cualquier carácter de cualquier lengua.

Se puede declarar e inicializar una variable de este tipo de la siguiente forma:

```
char unCaracter='a';
```

También se puede asignar a un tipo `char` una secuencia de escape. Por ejemplo:

```
char otroCaracter='\n';
```

En la tabla 3.7 se representan los caracteres de escape de C#.

Secuencia de escape	**Nombre del carácter**
`\'`	Comillas simples
`\"`	Comillas dobles
`\\`	Barra invertida
`\0`	Carácter Nulo
`\a`	Alerta
`\b`	Retroceso
`\f`	Avance de formulario
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulación horizontal
`\v`	Tabulación vertical

Tabla 3.7. Secuencias de escape.

No se pueden realizar conversiones implícitas de un tipo `char` a otros tipos. Es decir, no es posible tratar un `char` como si fuera un entero. Para trabajar de esta forma, tiene realizarse una conversión explícita. Por ejemplo:

```
char unCaracter=(char)65;
int unEntero=(int)'a';
```

Tipos enum

El tipo `enum` permite utilizar un grupo de constantes a través de nombres asociados que son más representativos que dichas constantes. Las constantes pueden ser de los tipos siguientes: `byte`, `short`, `int` o `long`. En su forma más simple una enumeración puede tener el siguiente aspecto:

```
enum Colores
{     Rojo=2,
      Verde=3,
      Azul=4
};
```

En ocasiones, se escribe en una sola línea:

```
enum Colores {Rojo=2,Verde=3,Azul=4};
```

Si no se especifica ningún valor, por ejemplo:

```
enum Colores
{     Rojo,
```

```
      Verde,
      Azul
};
```

El compilador asigna los valores por defecto siguientes : `Rojo` es `0`, `Verde 1` y `Azul 2` y son de tipo `int`. En general, el primer elemento es `0` y cada elemento sucesivo aumenta en una unidad.

Es posible cambiar el valor inicial y/o cualquier valor:

```
enum Colores
{
      Rojo=1,
      Verde,
      Azul
};
```

En este caso, `Rojo` es `1`, `Verde 2` y `Azul 3` y son de tipo `int`.

```
enum Colores2
{
      Rojo=20,
      Verde=30,
      Azul
};
```

`Rojo` es `20`, `Verde 30` y `Azul 31` y son de tipo `int`

Y también es posible cambiar el tipo:

```
enum Colores : byte
{
      Rojo,
      Verde,
      Azul
};
```

En este caso, `Rojo`, `Verde` y `Azul` serán de tipo `byte`.

Ejemplo:

```
using System;
public class MiClase{
      struct Dias {Lun, Mar=4, Mie, Jue=10, Vie, Sab,Dom};
      public static void Main(){
            int x=(int) Dias.Lun;
            int y=(int) Dias.Vie;

            Console.WriteLine("Lunes =  {0}, Viernes = {1}", x, y);
      }
}
```

La salida de este programa es:

```
Lunes = 0, Viernes = 11
```

Tipos struct

Un tipo `struct` es similar a un tipo `class` -puede contener constructores, constantes, métodos, etc...-, con una diferencia muy importante: `class` es de *tipo referencia*, y `struct` de *tipo valor*. Cuando se declara una variable de un tipo `class`, se crea el objeto con los datos en el *heap* o *montón* y una variable de referencia a ese objeto en la *pila* o *stack*. En cambio, cuando se declara una variable de un tipo `struct` los datos se almacenan directamente en la *pila*, ya que es un tipo valor. Por eso, el tipo `struct` es interesante para crear objetos "ligeros", es decir, de un tamaño pequeño, en los que una referencia significaría un incremento considerable en el tamaño de la variable. Para los cálculos en los que se utilicen únicamente los datos, es más rápido –en general- utilizar variables de tipo estructura. Sin embargo, cuando sea necesario pasarlo como parámetro a un método, se optimiza el trabajo si se pasan referencias a los objetos en lugar de los objetos propiamente dichos. Este concepto es interesante tenerlo en cuenta cuando se va a trabajar con matrices que contienen muchos objetos.

Las estructuras se estudiarán conjuntamente con las clases y aquí únicamente se introducirá un ejemplo:

```
using System;
struct Punto
{
   public int x, y;
   public Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
class MiClase{
      public static void Main(){
            Punto p=new Punto(7,9);
            Console.WriteLine("Coordenadas de p ({0},{1})", p.x, p.y);
      }
}
```

Tipos referencia.

Como se ha explicado en el apartado anterior, los tipos de referencia definen por un lado una referencia en la pila y por otro un objeto que se ha creado en el montón y que se asigna a la referencia (figura 3.1). Dicho de otro modo: una variable de tipo referencia es una referencia a una instancia u objeto de un determinado tipo. Las variables de tipo referencia no almacenan los datos que representan sino que almacenan referencias a dichos datos. Un tipo referencia admite el valor `null` que indica que no referencia o no apunta a ningún objeto.

Los tipos referencia implican la creación de:

a) una variable de referencia en la *pila*.
b) un objeto con los datos, que son apuntados por la variable de referencia en el *montón* (figura 3.1).

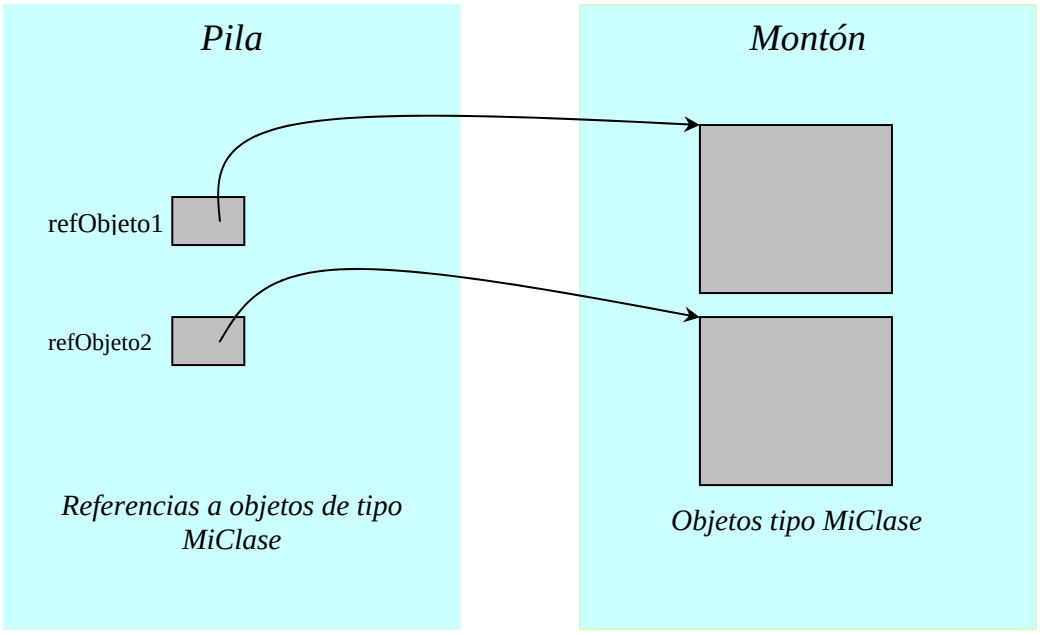


Figura 3.1

Existen los siguientes tipos de referencia:

- tipo clase
  - `class`
  - `object`
  - `string`
- tipo `interface`
- tipo `array`
- tipo `delegate`

Los tipos de referencia pueden ser predefinidos por el sistema o definidos por el programador.

Tipo class.

Un tipo `class` define una estructura de datos que puede contener miembros de datos (constantes, campos y eventos), funciones miembro (métodos, propiedades, indexadores, operadores, constructores y destructores) y otros tipos anidados.

Los tipos `class` –las clases- soportan la herencia, pero sólo simple, excepto en el caso de la herencia de interfaces que es la única que puede ser múltiple.

Más adelante se verán en detalle las clases. Tienen la misma composición que las estructuras.

Tipo object.

Como se ha mencionado anteriormente, todos los tipos de C# heredan del tipo `System.Object` y se puede decir que en C# todos los tipos de datos son objetos. Esto garantiza que cualquier tipo tenga los métodos de la clase `object`.

La palabra clave `object` es un alias de la clase predefinida `System.Object`.

Una característica del tipo `object` es que una variable de este tipo admite cualquier valor, por ejemplo:

```
object refObjeto = 20;
```

La sentencia anterior es –en realidad- una conversión implícita de un dato de *tipo valor* a uno de *tipo referencia*, es decir, un *boxing* o encuadramiento. Este importante concepto se estudiará en detalle más adelante, en este mismo capítulo.

En la tabla 3.8, se describen los métodos públicos -`public`- más importantes de la clase `System.Object` y que cualquier variable puede utilizar por el hecho de ser objeto.

`bool Equals(object oA)`	Compara los objetos. Devuelve `True` o `False` dependiendo de que sean o no el mismo objeto. En el caso de variables de tipo valor este método devuelve `True` si los dos tipos son idénticos y tienen exactamente el mismo valor.
`Type GetType()`	Se usa para métodos de reflexión y devuelve el tipo de un objeto dado.
`string ToString()`	Devuelve el nombre del objeto. Se puede sobrescribir.

Tabla 3.8. Algunos métodos de la clase `System.Object`.

En los siguientes ejemplos se puede observar cómo cualquier variable de cualquier tipo puede utilizar los miembros de `object`.

```
int a=56;
Object oA=a;
Console.WriteLine(a.ToString());
Console.WriteLine(a.GetType());
Console.WriteLine(56.ToString());
Console.WriteLine(a.Equals(56));
Console.WriteLine(oA.ToString());
```

La salida de este programa es:

```
56
Int32
56
True
56
56
```

Es importante señalar aquí que cuando se comparan variables de tipos predefinidos, hay que tener en cuenta los siguiente:

- Dos expresiones de tipo entero, real, etc, son iguales si representan el mismo valor entero, real, etc.
- Dos expresiones de tipo `object` se consideran iguales si ambas se refieren al mismo objeto o si ambas son `null`.
- Dos expresiones de tipo `string` se consideran iguales si las instancias –los objetos- tienen idéntica longitud y caracteres iguales en las mismas posiciones o si ambos son `null`.

Tipo string.

El tipo `string` representa una cadena de caracteres UNICODE. `string` es un alias de la clase `System.String`. El tipo `string` hereda o deriva directamente del tipo `object`.

Un objeto `string` se maneja como un objeto de una clase cualquiera. Pero además, en este caso, se permite crearlo y utilizarlo directamente, a partir de literales:

```
string unaCadena = “Hola”;
string otraCadena = “Mundo”;
```

Esta clase tiene muchos constructores. Algunos de ellos se describen a continuación:

a) `String(char unCaracter, int numero);`
Construye un `string` formado por el carácter `unCaracter`, repetido tantas veces como indica `numero`.

b) `String(char[] unArray);`
Construye un string con la cadena formada por los caracteres del array unArray

c) `String(char[] miArray, int comienzo, int longitud);`
Construye un string basado en la cadena formada por los caracteres de miArray, pero comenzando en el carácter comienzo y a partir de él tantos caracteres como indique longitud.

d) Por supuesto, se puede construir un `string` asignándole un literal como se ha hecho anteriormente.

En el siguiente ejemplo se utilizan varios constructores diferentes:

```
string str1=new string('m',7);
Console.WriteLine(str1);
char[] miArray={'h','o','l','a'};
string str2=new string(miArray);
Console.WriteLine(str2);
string str3=new string(miArray,1,2);
Console.WriteLine(str3);
string str4="Ian es un paquete";
Console.WriteLine(str4);
```

La salida de este programa es:

```
mmmmmmm
hola
ol
Ian es un paquete
```

Por alguna extraña razón que los autores desconocen, Microsoft ha dispuesto que un `string` se construya como si fuera un tipo valor, aunque realmente no lo es. Es decir, no se permite la expresión:

```
string str5=new string("Hola Mundo"); //ERROR
```

A continuación, se describen algunas características del tipo `string`:

- Se pueden concatenar dos cadenas con el operador + , que está sobrecargado:

Por ejemplo:

```
string unaCadena = “Hola”;
string otraCadena = “Mundo”;
Console.WriteLine(unaCadena + otraCadena);  //La salida es HolaMundo

string s1=”Una cadena” + “otra cadena”;
Console.WriteLine(s1);        //La salida es: Una cadenaotra cadena
```

- El operador `[]` accede individualmente a los caracteres del `string`:

```
string str=”Miguel”
char c1=str[3];         //c1 = ’u’
char c2=”Patricia”[2]   //c2=’t’
```

- Aunque un `string` es de *tipo referencia,* los operadores sobrecargados == y ¡= comparan los valores de los objetos `string` no las referencias a los objetos.

```
string s1=”Hola”;
string s2=”Hola”;
Console.WriteLine(s1==s2);                    //La salida es True
Console.WriteLine((object)s1==(object)s2);    //La salida es False
```

Esto es así porque `s1` y `s2` son dos referencias a dos objetos distintos –físicamente están almacenados en posiciones de memoria diferentes-, pero que tienen el mismo valor. En la primera sentencia, se comparan los valores de los objetos –que son iguales- y en la segunda, las referencias -que son distintas-.

- El contenido de un objeto `string` es constante y no se puede cambiar.

Por ejemplo:

```
string unaFrase=”Esta es una frase”;

unaFrase+=” ingeniosa”;
```

Cuando este código se ejecuta ocurre lo siguiente: en primer lugar, se crea un objeto de tipo `System.String` y se inicializa con el texto "`Esta es una frase`". .NET reserva memoria en el *montón* suficiente para albergar este objeto –17 caracteres- y crea una referencia `unaFrase` en la pila de tipo `string` que apunte a ese objeto.

En la línea siguiente parece que se añade un texto en el `string` anterior, en el objeto. Pero esto no sucede así. Lo que realmente ocurre es que se crea un nuevo objeto y se reserva la memoria suficiente para almacenar un nuevo objeto de tipo `string` con ese nuevo contenido –27 caracteres-. Entonces la referencia de tipo `string unaFrase` se actualiza para que referencie o apunte al nuevo objeto. El antiguo objeto `string` sigue existiendo en memoria, pero es desrreferenciado y será borrado por el *garbage collector* (recolector de basura).

Para poder tratar los `string` como se supone que deberían ser tratados –es decir, no siendo constantes- .NET proporciona otra clase, llamada `StringBuilder`. Se utiliza para situaciones en las cuales se desea modificar un `string`, añadiendo, reeemplazando o insertando caracteres, sin crear un nuevo objeto `string` en cada modificación. Los métodos de esta clase no devuelven un nuevo objeto `StringBuilder` a menos que se especifique otra cosa.

- Los literales de tipo `string` se pueden escribir de dos formas: precedidos o no por el carácter `@`. Si no se precede de `@`, se define con comillas dobles. Por ejemplo:

```
"Buenos días"  // un literal de tipo string
```

La otra forma es escribir el literal de `string` comenzando con el carácter "arroba" `@` y encerrándolo entre comillas dobles. Por ejemplo:

```
@"Buenos días"  // un literal
```

La ventaja de esta forma de escribir un literal es que no se procesan las secuencias de escape. Esto hace que sea más sencillo escribir, por ejemplo, el path de un determinado fichero:

```
@"c:\Docs\Source\a.txt"
```

Es mejor que escribir:

```
"c:\\Docs\\Source\\a.txt"
```

Para incluir las comillas dobles en la notación precedida por `@`, hay que escribirlas dos veces. Por ejemplo:

```
@"""¡Hola!"" le gritó desde lejos." // "¡Hola!" le gritó desde lejos.
```

- Esta clase tiene un buen número de métodos. A continuación se explican algunos de ellos que facilitan enormemente el trabajo con cadenas.

  ```
  -int IndexOf(char unCaracter);
  ```

Devuelve un entero con el lugar de la primera ocurrencia de `unCaracter`, en el `string` que llama al método.

```
int IndexOf(char unCaracter,int comienzo);
```

Devuelve un entero con el lugar de la primera ocurrencia de `unCaracter`, a partir de `comienzo`, en el `string` que llama al método.

-`int IndexOf(char unCaracter,int comienzo, int longitud);`

Devuelve un entero con el lugar de la primera ocurrencia de `unCaracter`, y examinando `longitud` caracteres a partir de `comienzo`.

En todos los casos, si no se encuentra ningún carácter coincidente, devuelve `-1`. Por ejemplo:

```
Console.WriteLine("Hola MundoMundial".IndexOf('M'));
Console.WriteLine("Hola MundoMundial".IndexOf('M',7));
Console.WriteLine("Hola MundoMundial".IndexOf('M',7,5));
```

La salida es:

```
5
10
10
```

-`int Length`

Devuelve un entero con la longitud de la cadena. Es una propiedad, no un método.

```
Console.WriteLine("El Madrid es el mejor".Length);   //21
```

-`string Insert(int comienzo, string unaCadena);`

Devuelve la cadena actual donde se le ha insertado a partir de `comienzo` el `string` `unaCadena`.

```
Console.WriteLine("Hola".Insert(2,"juan")); //HoJuanla
```

-`string Remove(int comienzo, int numCaracteres);`

Devuelve el `string` resultante de borrar `numCaracteres` desde `comienzo` del `string` actual.

```
Console.WriteLine("El Madrid es el mejor".Remove(4,8));
```

-`string Replace(string viejo, string nuevo);`

Devuelve una cadena con el `string` actual donde se ha sustituido todas las ocurrencias de `viejo` por `nuevo`. Por ejemplo:

```
Console.WriteLine("El Madrid es el mejor".Replace("Madrid",
                  "Sevilla"));
```

La salida es:

```
El Sevilla es el mejor
```

`-string ToUpper() // string ToLower()`

Devuelve el `string` actual en mayúsculas o minúsculas. Por ejemplo:

```
Console.WriteLine("El Madrid es el mejor".ToUpper());
```

La salida es:

```
EL MADRID ES EL MEJOR
```

A continuación se resumen otros métodos útiles:

`StartsWith`	Determina si el `string` comienza por un `string` especificado.
`EndsWith`	Determina si el `string` finaliza por un `string` especificado
`Substring`	Sobrecargado. Devuelve un `substring` desde el `string` actual.
`ToCharArray`	Sobrecargado. Copia los caracteres a un array de caracteres.
`Trim`	Sobrecargado. Borra un conjunto de caracteres especificados al comienzo y al final de un `string`. Muy utilizado en programación visual.
`TrimEnd`	Borra un conjunto de caracteres especificados al final de un `string`.
`TrimStart`	Borra un conjunto de caracteres especificados al comienzo de un `string`.

Tabla 3.9

Tipo interface.

Un interface es similar a una clase o a una estructura con la salvedad de que todos sus miembros son abstractos, es decir, no están definidos, no tienen código.

Las interfaces están pensadas para representar un aspecto de la realidad que habrá de ser *implementado* por una clase que herede la interface.

Ejemplo:

```
interface IVolar
{
      void Volar();
      void Despegar();
      void Aterrizar();
}


class CAvion:IVolar
{
      public void Volar()
      {
            //Código que implementa el método IVolar()
            ...
      }
```

```
	public void Despegar()
	{
		//Código que implementa el método IDespegar()
		...
	}
	public void Aterrizar()
	{
		//Código que implementa el método IAterrizar()
		...
	}
}
```

No es posible crear objetos de tipo `interface`, porque sus métodos no están implementados, pero sí es posible referenciar a un objeto de una clase que implementa cierta interface a través de una referencia a esa interface. Por ejemplo:

```
IVolar iv = new CAvion();
```

o

```
CAvion ca = new CAvion();
IVolar iv = (IVolar) ca;
```

Tipo Array.

Un array es un tipo referencia que contiene un conjunto ordenado de datos que son variables (elementos) a los que se accede a través de índices. Estas variables pueden ser –a su vez- de tipo valor o de tipo referencia.

Todas las variables de un array son del mismo tipo.

Para **declarar** un array, se escribe el tipo de elementos del array, después el operador `[]` y, por último, el nombre del array. Por ejemplo:

```
int[] unArray;
```

En la anterior sentencia se ha declarado un `array` de enteros de nombre `unArray`. `unArray` es una referencia al array.

Se puede crear un objeto de tipo array con el operador `new`.

```
int[] miArray=new int[4];     //Se crea un objeto de tipo array
                              //de 5 enteros inicializados a 0
```

Se puede también *inicializar* los elementos del array:

```
int[] miArray=new int[4]{2,5,12,56,3};
```

o mejor:

```
int[] miArray=new int[]{2,5,12,56,3};
```

o bien, se puede compactar más la declaración y definición e inicialización del array:

```
int[] miArray={2,5,12,56,3};
```

El tipo `array` tiene una serie de métodos que se estudiarán más adelante. Sin embargo, se quiere destacar aquí la propiedad `Length`, que será muy útil para recorrer el `array`. Esta propiedad devuelve un entero con el número de elementos del `array`.

En el ejemplo anterior:

```
Console.WriteLine(miArray.Length); //Imprime 5
```

Un array puede tener más de una dimensión. Por cada dimensión de un array se necesitará un nivel de índices para acceder a sus elementos.

Ejemplo:

```
string[3,2] matrizStrings;
int[,] otroArray = {{3,5,7},{6,89,21},{34,56,7}};
```

Tipo Delegate.

Un `delegate` es muy similar a un puntero a función de C++. Es una estructura de datos que referencia a un método estático o a un método de instancia de un objeto.

Existen algunas diferencias:

- Un puntero a función de C++ sólo puede referenciar funciones estáticas
- Un delegate no sólo referencia al punto de entrada del método, sino también a la instancia del objeto al que pertenece el método.

Tipo puntero.

Las variables de este tipo son los punteros tradicionales de C y C++. Si se utilizan, al código donde son utilizados se le denomina código inseguro.

Boxing y UnBoxing.

Una vez estudiados los tipos de datos en C#, se plantean dos cuestiones importantes:

a) ¿Cómo estas diferentes categorías de tipo hacen que el sistema sea más eficiente?.
b) ¿Se puede utilizar una variable de tipo valor como si fuera de tipo referencia – es decir, como si fuera un objeto- y viceversa?.

Para responder a ambas cuestiones es necesario comprender los conceptos **boxing** y **unboxing**. Estos dos procesos son muy importantes porque proporcionan un vínculo entre los tipos valor y los tipos referencia y permiten la conversión entre ellos.

Boxing es la conversión de un *tipo valor* a un *tipo referencia*. *Unboxing* es el proceso inverso, que se utiliza para convertir un objeto que contiene un valor, al tipo de valor correspondiente.

Como anteriormente se ha dicho, todo tipo en C# deriva directa o indirectamente del tipo `object`, que es la clase raíz de la jerarquía de herencia en C#. Por eso, se puede

tratar un tipo valor bien como tipo valor o bien como objeto (como tipo referencia) indistintamente.

Cuando sea preciso "convertir" una variable tipo valor al tipo referencia (es decir a un objeto, al tipo `object`) se realiza un **boxing** o encuadramiento. Así, por ejemplo, un `int` puede ser tratado como un tipo valor o como un tipo referencia (como un objeto).

Por eso es correcto escribir:

```
Console.WriteLine(7.ToString());
```

Realmente, aquí, `7` es un objeto y puede invocar el método `ToString()` de `object`

En general, el *boxing* o encuadramiento es una operación se realiza de *modo implícito* y consiste en crear un objeto a partir de un tipo valor. Ejemplo:

```
int unValor = 100;                  //variable de tipo valor
object objValor = unValor;          //referencia a un objeto
```

La segunda línea realiza implícitamente el *boxing* ya que se "convierte" –de manera implícita- un *tipo valor* a un objeto, que es de tipo referencia.

Para comprender mejor este proceso, debe tenerse en cuenta que tanto la variable de tipo valor `unValor` como la referencia al objeto `objValor` se almacenan en la *pila (stack)*. El objeto se almacena en el *montón* (*heap*).

Si se muestra gráficamente el ejemplo anterior paso a paso, la situación será:

1) `int unValor = 100;`

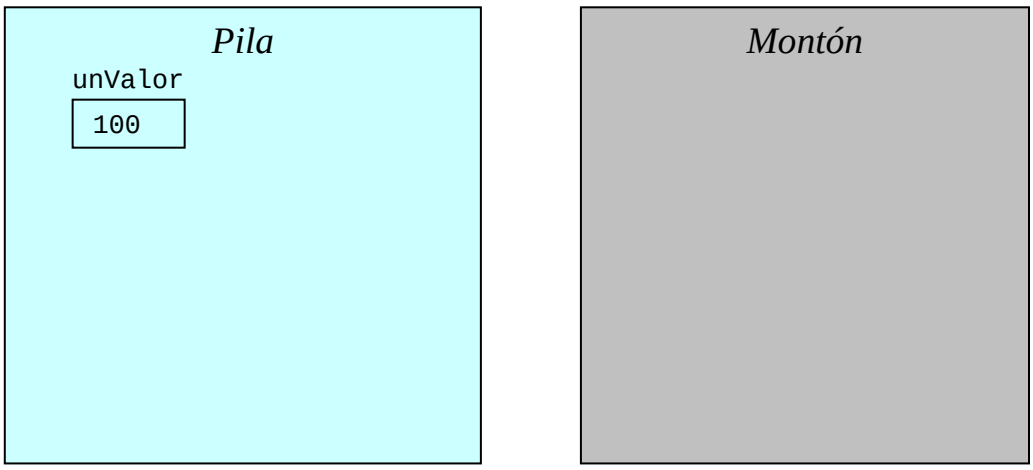


Figura 3.2

2) `object objValor = unValor;`

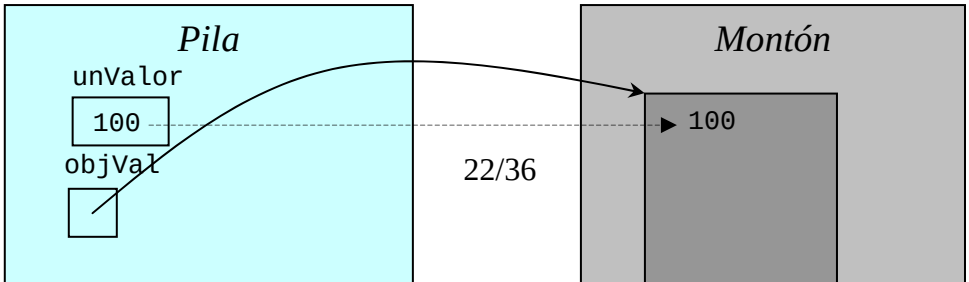


Figura 3.3

Como se puede apreciar, la operación **boxing** implica crear un objeto que es una copia en el montón o heap de la variable tipo valor, pero manteniendo la variable. Por eso, cualquier cambio que se haga sobre `objValor` no afectará a `unValor`.

Unboxing.

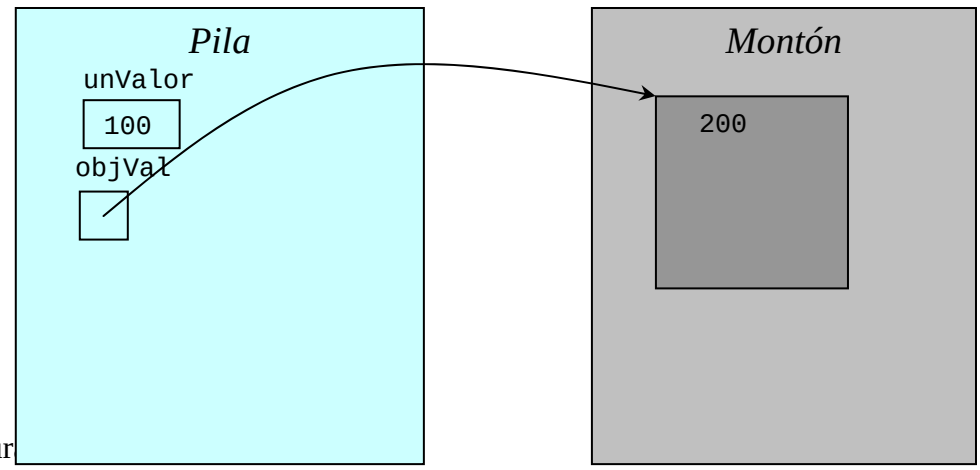
Es una operación explícita y consiste en asignar a una variable de tipo valor el valor de una instancia u objeto referenciada por una variable de tipo referencia.

Ejemplo:

```
int unValor=100;
object objValor= 200;
unValor = (int) objValor;     //el casting o conversión ha de ser
                              //explícito.
```

A continuación se muestra el ejemplo paso a paso:

```
1)    int unValor=100;        //Variable de tipo valor
      objValor = 200;         //Variable de tipo referencia
```



Figur

```
2) unValor=(int)objValor;     //la conversión (unboxing) debe ser
                              //explícita.
```

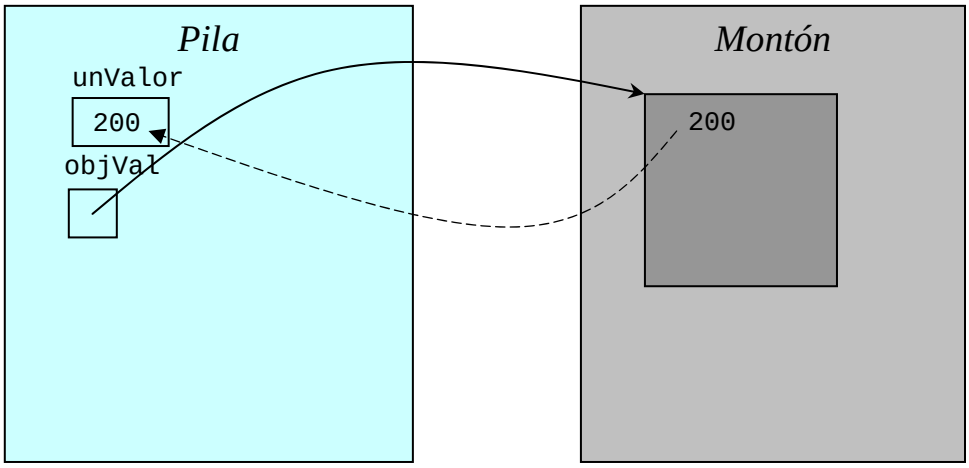


Figura 3.5

Como se puede observar, el casting o conversión ha de ser explícito.
Además, aunque se indique explícitamente, si el casting o conversión no es al tipo valor correcto el Runtime de la plataforma .NET lanzará una excepción de la clase `InvalidCastException` en tiempo de ejecución.

Variables y Expresiones.

Variables.

Las variables representan zonas de memoria donde se almacenan datos. Toda variable está asociada a un tipo que determina qué es lo que puede almacenar.

Puede ser utilizada desde el código mediante un identificador -`A`, `var1`, `x_3`, etc...-. Para *declarar* una variable sólo hay que *definir* un nombre y *especificar* el tipo. La sintaxis es:

```
Tipo NombreDeLaVariable;
```

Por ejemplo:

```
int unEntero;
object unObjeto;
```

Una variable puede ser definida

a) Como miembro de una clase
b) Como variable local
c) Como parámetro de un método

Si una variable es declarada pero no *inicializada*, el compilador lo hará por nosotros, con unos valores –que dependen del tipo- preestablecidos, como se ha estudiado previamente.

Para crear una variable, se debe distinguir entre tipos valor y tipos referencia. Para crear un objeto de una clase, se utiliza la siguiente sintaxis:

```
NombreDeClase nombreDeLaReferencia = new NombreDeClase(Parámetros_opt);
```

Por ejemplo: Suponiendo que en un determinado lugar del programa se ha definido la clase `Rectángulo`, se puede crear un objeto de esa clase del siguiente modo:

```
Rectángulo unRectangulo=new Rectángulo(4,9);
```

Esta sentencia crea un objeto de tipo `Rectángulo`, inicializado con los valores `4` y `9`. El objeto se almacena en el montón y la referencia en la pila.

En las variables de tipo valor y en algunas de tipo referencia se puede crear e inicializar de manera simultánea el objeto o variable de la siguiente forma:

```
int unEntero=5;
string unaCadena=”Hola Mundo”;
object unObjeto=45;
MiClase unaClase=null;
```

El lenguaje C# es de “tipos-seguros”, lo que significa que el compilador garantiza que los valores almacenados en las variables son del tipo adecuado.

El valor de una variable puede ser modificado a través de la operación de asignación o mediante el uso de los operadores de incremento y decremento (`++`, `--`).

En C# se definen siete categorías de variables:

- variables `static`.
- variables de instancia.
- elementos de array.
- parámetros valor.
- parámetros `ref` o de referencia.
- parámetros `out` o de salida.
- variables locales.

Las variables `static`, de instancia y los elementos de array son automáticamente inicializadas al valor por defecto del tipo al que pertenecen.

Variables static o variables de clase.

Se llama variable `static` a cualquier campo declarado con el modificador `static`. Las variables `static` se crean cuando el tipo en el que son declaradas se carga. El valor inicial de una variable `static` es el valor por defecto del tipo al que pertenece.

Por ejemplo:

```
class Prueba
{
   public static int x;
   ...
   ...
}
```

Su uso podría ser:

```
Prueba.x = 5;
Prueba p1 = new Prueba();
Prueba p2 = new Prueba();

p1.x = 10;
```

Lo relevante de las variables estáticas -`static`- es que sólo hay una para todas las instancias u objetos que se creen del tipo que las contiene, no una por cada objeto que se cree. Por eso se les llama también *variables de clase* y pueden invocarse a través del nombre de la clase: no es necesario crear un objeto de la clase. Tras el ejemplo anterior, tanto `p1.x` como `p2.x` valdrán `10`, ya que sólo existirá una variable `x` para ambos. Más adelante, al hablar del concepto de clase, se ampliará este concepto.

Variables de instancia.

Cualquier campo declarado sin el modificador `static` se considera una variable de instancia.

Una variable de instancia puede pertenecer a un tipo `class` o a un tipo `struct`.
Cada objeto de un determinado tipo tendrá sus propias variables de instancia.

Por ejemplo:

```
class Prueba
{
   public int x;
   ...
   ...
}
```

Su uso podría ser:

```
Prueba p1 = new Prueba();
Prueba p2 = new Prueba();

p1.x = 10;
p2.x = 20;
```

En este ejemplo, la variable `x` de la instancia referenciada por `p1` valdrá `10` y la de la instancia referenciada por `p2` valdrá `20`.

No tiene sentido lo que se indica a continuación, ya que `x` es una variable que no está asociada a cada objeto y existe una variable `x` distinta para cada objeto:

```
Prueba.x = 5; //esta instrucción dará un error de compilación.
```

Al estudiar los modificadores de acceso, se tratarán con más profundidad los conceptos de variable estática o de clase y campo o variable de objeto o instancia

Elementos de Array.

Los elementos de un array son creados cuando se crea la instancia del array y se desasignan cuando la instancia del array deja de ser referenciada. El valor inicial de cada uno de los elementos del array es el valor por defecto del tipo de los elementos del array.
Ejemplo:

```
int[] arrNumerosEnteros = {5, 7 ,9};
int[5] arrNumerosEnteros2;
string[3,5] matrizStrings;
...
arrNumerosEnteros[0] = 20;
arrNumerosEnteros2[0] = 25;
matrizStrings[2,2] = “Hola”
...
```

Existe una clase `System.Array` en el Framework .NET que encapsula el manejo de los arrays tradicionales.

Parámetros.

Pueden ser de tres tipos:

- **Parámetros valor**: son los parámetros que no tienen modificador. Un parámetro valor se crea para pasar una variable por valor a un método. Se crea una copia del valor del parámetro y se destruye cuando la función acaba y devuelve el control (es similar a una variable local).

  Ejemplo:

```
// Suma dos números
using System;
public class ClaseSumadora
{
   public static long Añadir(long i, long j)
   {
      return(i+j);
   }
}
```

  La llamada es de la forma:

```
ClaseSumadora. Añadir(20, 50);
```

- **Parámetros referencia**: se declaran utilizando el modificador `ref`. Un parámetro referencia representa la referencia a una instancia existente de un tipo.

  Ejemplos:

```
class Prueba
{
   static void F(ref object x) {...}
   static void Main() {
      object[] a = new object[10];
      F(ref a[0]);
   }
}

class Prueba2
{
   static void Swap(ref int x, ref int y) {
      int temp = x;
      x = y;
      y = temp;
   }
   static void Main() {
      int i = 1, j = 2;
      Swap(ref i, ref j);
      Console.WriteLine("i = {0}, j = {1}", i, j);
   }
}
```

- Parámetros `out` (de salida): se declaran utilizando el modificador `out`. Un parámetro de tipo `out` o salida es realmente una referencia a una instancia ya existente. La

diferencia con un parámetro referencia es que la instancia referenciada por el parámetro `out` no tiene porqué estar inicializada.

```
class Prueba3
{
   static void Salida(out int x) {
      x = 5;
   }
   static void Main() {
      int i;
      Salida(out i);
      Console.WriteLine("i = ", i);
   }
}
```

Variables Locales.

Una variable local puede ser declarada dentro de un bloque, una sentencia `for` o una sentencia `switch`, etc. El ciclo de vida de una variable local es el bloque o sentencia en el que ha sido declarado.

Una variable local no es inicializada automáticamente y no tiene valor por defecto, por lo tanto ha de ser inicializada explícitamente.

En el ejemplo siguiente, la variable `i` es local a `for`:

```
class MiClase
{
      static void Main(string[] args)
      {
            for(int i=0;i<3;i++)
            {
                  Console.Write("Elemento {0}",i);
                  Console.WriteLine("{0}",i*i);
            }
      }

  }
```

Operadores.

Existen tres tipos de operadores:

- Unarios: Se asocian a un solo operando. Algunos pueden ir delante (`++x`, `-x`) y otros detrás del operando (`x++`).
- Binarios: Se asocian a dos operandos. Por ejemplo: `x+y`.
- Ternarios: se asocian a tres operandos. Sólo existe un operador ternario, `?:`. Por ejemplo: `var1? valor2: valor3`.

El orden de evaluación de los operadores en una expresión está determinado por la precedencia y asociatividad de los operadores.

La tabla 3.10 muestra la precedencia de los operadores. Está ordenada de mayor a menor precedencia:

Categoría	**Operadores**
Primarios	`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked`
Unarios	`+ - ! ~ ++x --x (T)x`
Multiplicativos	`* / %`
Aditivos	`+ -`
Desplazamiento	`<< >>`
Relacionales	`< > <= >= is`
Igualdad	`== !=`
AND Lógico	`&`
XOR Lógico	`^`
OR Lógico	`|`
AND Condicional	`&&`
OR Condicional	`||`
Condicionales	`?:`
Asignación	`= *= /= %= += -= <<= >>= &= ^= |=`

Tabla 3.10. Precedencia de los operandos.

Cuando un operando se encuentra entre dos operadores con la misma precedencia, la asociatividad de los operadores controla el orden en el cual se realizan las operaciones.

- Excepto los operadores de asignación, todos los operadores binarios son asociativos a izquierdas, es decir, la expresión:

  ```
  x + y + z
  ```

  se evalúa como

  ```
  (x + y) + z
  ```

- El operador de asignación y el operador condicional son asociativos a derechas. Por ejemplo:

  ```
  x = y = z
  ```

  se evalúa como

  ```
  x = (y = z).
  ```

- La precedencia y asociatividad pueden ser modificadas mediante el uso de paréntesis.

  Por ejemplo:

  ```
  (x = y) = z
  ```

se evalúa de izquierda a derecha. El orden preestablecido sin paréntesis es de derecha a izquierda.

Operadores de información de Tipo.

typeof.

El operador `typeof` se utiliza para obtener el objeto `System.Type` correspondiente a un tipo. Se utiliza mucho en la reflexión, cuando se quiere obtener información acerca de un determinado tipo dinámicamente. La sintaxis es:

```
Type typeof(tipo)
```

donde:

tipo
Es el tipo para el cual se quiere obtener el objeto `System.Type`.

`typeof` devuelve un objeto de tipo `Type`.

Ejemplo:

```
Type t1=typeof(MiClase);
Type t2=typeof(int);
```

Para obtener el tipo de una determinada expresión, objeto, o instancia, C# proporciona el método `GetType()`, cuya estructura es:

```
System.Type unObjeto.GetType();
```

`GetType()` es un método que todos los tipos heredan de la clase `object`. Devuelve un objeto del tipo `Type` del objeto `unObjeto`.

Por ejemplo:

```
double unDouble=98.34;
Type t=unDouble.GetType();
```

Un ejemplo completo:

```
using System;

namespace PruebaNamespace
{

      class MiClase
      {
            int unEntero;
            public double unDouble;
            static void Main(string[] args)
            {
                  MiClase unaClase=new MiClase();
                  Type t1=typeof(MiClase);
```

```
                  Console.WriteLine(t1.ToString());
                  Type t2=typeof(int);
                  Console.WriteLine(t2.ToString());
                  Type t3=unaClase.unEntero.GetType();
                  Console.WriteLine(t3.ToString());
                  //unaClase.unDouble.
                  Type t4=unaClase.unDouble.GetType();
                  Console.WriteLine(t4.ToString());


            }
      }
}
```

La salida de este programa es:

```
PruebaNamespace.MiClase
System.Int32
System.Int32
System.Double
```

is.

El operador `is` se utiliza para chequear en tiempo de ejecución si el tipo de una expresión es compatible con un tipo de objeto dado.

expresion **is** *tipo*

donde:

expresion
: Es una expresión de tipo referencia.

tipo
: Es un tipo referencia.

Este operador no puede ser sobrecargado. Devuelve `True` o `False`.

Por ejemplo:

```
using System;

namespace PruebaNamespace
{

      class MiClase
      {
            static void Main(string[] args)
            {
                  int unEntero=7;
                  MiClase unaClase=new MiClase();
                  Console.WriteLine(3*5 is int);
                  Console.WriteLine(unaClase is MiClase);
                  if(unEntero is object)
                        Console.WriteLine("unEntero es un objeto");
            }
      }
}
```

La salida es

```
True
True
unEntero es un objeto
```

as.

El operador `as` se utiliza para realizar conversiones entre tipos compatibles. Se utiliza del siguiente modo:

expresion **as** *tipo*

donde:

expresion
: Es una expresión de tipo referencia.

tipo
: Es un tipo referencia.

El operador `as` realiza una conversión. Si dicha conversión no se realiza con éxito, se devuelve el resultado de la conversión. En caso contrario se devuelve `null` pero sin lanzar una excepción.

En realidad, la expresión:

`expresion` **as** *tipo*

es equivalente a:

`expresion is` *tipo*`? (tipo)expresion : (tipo)null`

con la diferencia de que la expresión se evalúa sólo una vez.

Ejemplo:

```
using System;

namespace PruebaNamespace
{
      class MiClase1
      {
      }

      class MiClase2
      {
      }

      class MiClase
      {
            public static void Main()
            {
```

```
                object [] misObjetos = new object[6];
                misObjetos[0] = new MiClase1();
                misObjetos[1] = new MiClase2();
                misObjetos[2] = "hola";
                misObjetos[3] = 123;
                misObjetos[4] = 123.4;
                misObjetos[5] = null;

                for (int i=0; i< misObjetos.Length; ++i)
                {
                      string s = misObjetos[i] as string;
                      Console.Write ("{0}:", i);
                      if (s != null)
                            Console.WriteLine ( "'" + s + "'" );
                      else
                            Console.WriteLine ( "no es un string" );
                }
          }

     }
}
```

La salida de este programa es:

```
0:no es un string
1:no es un string
2:'hola'
3:no es un string
4:no es un string
5:no es un string
```

Sobrecarga de operadores.

Existe una implementación predefinida para todos los operadores unarios y binarios. No obstante, en clases y estructuras es posible sobrecargar ciertos operadores. Un operador sobrecargado siempre tiene preferencia sobre el operador predefinido.

Los operadores unarios que es posible sobrecargar son:

```
+   -   !   ~   ++   --   True   False
```

Los binarios son:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

No es posible sobrecargar:

- el acceso a miembros de un tipo.
- la invocación de métodos.
- los operadores `=, &&, ||, ?:, new, typeof, sizeof e is.`

Cuando se sobrecarga un operador binario, el correspondiente operador de asignación es sobrecargado implícitamente. Por ejemplo, una sobrecarga del operador + es también una sobrecarga del operador +=.

Más adelante, cuando se hable de la sobrecarga de métodos, se tratará con más detalle la sobrecarga de operadores.

Conversiones.

Conversiones implícitas.

Cualquier tipo puede convertirse de modo implícito a otro tipo según el orden que se adjunta a continuación:

- `sbyte`→ `short` → `int` → `long` → `float` → `double` o `decimal`

- `byte` → `short` → `ushort`→ `int` → `uint` → `long` → `ulong` → `float` → `double` o `decimal`

- `char` → `ushort`…

En todos esos casos no se pierden datos.

Además se pueden realizar las siguientes conversiones implícitas:

- conversiones implícitas de enumeraciones: permiten convertir un literal entero decimal a un tipo `enum`.
- conversiones implícitas de referencia (estándar):
    - De cualquier tipo referencia a `object`.
    - De cualquier tipo `class` a otro tipo `class` que sea superclase suya.
    - De cualquier tipo `class` a cualquier tipo `interface`, si el primero implementa la `interface`.
    - De cualquier tipo `array` a `System.Array`.
    - De cualquier tipo `delegate` a `System.Delegate`.
    - De cualquier tipo `array` o tipo `delegate` a `System.ICloneable`.
    - De tipo `null` a cualquier tipo referencia.
- Boxing (estándar).
- Conversiones implícitas de constantes (estándar):
    - Una expresión *constante* de tipo `int` puede ser convertida a tipo `sbyte`, `byte`, `short`, `ushort`, `uint`, o `ulong`.
    - Una expresión `constant` de tipo `long` puede ser convertida a tipo `ulong`.
- Conversiones implícitas definidas por el usuario.

Conversiones explícitas.

Las conversiones explícitas han de ser indicadas explícitamente y se pueden aplicar a:

- Todas las conversiones implícitas.
- Unboxing.

- conversiones numéricas.
- conversiones de referencia.
- conversiones de enumeraciones.
- conversiones de interfaces.
- conversiones explícitas definidas para el usuario.