6 DEFINIENDO NUESTROS TIPOS

En los capítulos anteriores se han utilizados los tipos básicos integrados en el lenguaje, como `int`, `bool`, `double`, `string`, así como otros tipos disponibles en la biblioteca de .NET Framework, como `Console`, `Math`, `Stopwatch`.

Por muy amplia y abarcadora que pueda ser la biblioteca de .NET Framework, no tiene sentido pretender que sea suficiente para todas las necesidades que nuestras aplicaciones puedan tener. Precisamente una de las características más importantes que tiene la Programación Orientada a Objetos (en lo adelante POO), y de los lenguajes que sustentan esta programación como es el caso de C#, es que disponen de recursos para que los desarrolladores puedan definir sus propios tipos.

6.1 Clases

El concepto de **clase** es un mecanismo para definir tipos. Tipos como `Console`, `Math` y `Stopwatch` están definidos por clases.

El Listado 6.1 nos muestra un bosquejo de la clase `Console` con los métodos `ReadLine` y `WriteLine`.

```
namespace System
{
  public static class Console
  {
    public static string ReadLine();
    public static void WriteLine(string format, params object[] arg);
    // ...
  }
}
```

Listado 6.1Esbozo de la clase Console

Aquí el recurso `class` indica que se está definiendo un tipo. La definición de un tipo mediante una clase tiene la sintaxis

```
class <nombre de la clase>
{
  <métodos, propiedades, variables y otros recursos de la clase>
}
```

Que los métodos `ReadLine` y `WriteLine` estén precedidos por la especificación `public` los hace "visibles" hacia fuera de la clase, es decir que pueden ser invocados desde el código de otros métodos que no sean de la clase. Que además estén precedidos por la especificación `static` hace que estos métodos puedan ser invocados a través del nombre de la clase en sentencias como

```
string s = Console.ReadLine();
Console.WriteLine("El MCD de {0} y {1} es {2}", m, n, Mcd(m,n));
```

Realmente la clase `Console` tiene una gran cantidad de métodos que no podemos ahora detallar aquí. El lector puede familiarizarse con ellos a través de la documentación disponible en el propio Visual Studio.

Es decir, para invocar al método se ha utilizado en este caso la notación

<nombre del tipo>.<nombre del método>(<parámetros del método>)

Los recursos de la clase `Stopwatch` se muestran en el Listado 6.2

```
1  namespace System.Diagnostics
2  {
3    public class Stopwatch
4    {
5      public static readonly long Frequency;
6      public static readonly bool IsHighResolution;
7      public Stopwatch();
8      public TimeSpan Elapsed { get; }
9      public long ElapsedMilliseconds { get; }
10     public long ElapsedTicks { get; }
11     public bool IsRunning { get; }
12     public static long GetTimestamp();
13     public void Reset();
14     public void Restart();
15     public void Start();
16     public static Stopwatch StartNew();
17     public void Stop();
18   }
19 }
```

Listado 6.2 Recursos de la clase Stopwatch

En los ejemplos de capítulos precedentes se han utilizado algunos de estos recursos, como las propiedades `EllapsedMilliseconds` e `IsRunning` y los métodos `Restart`, `Start` y `Stop`. Note que a diferencia del método `WriteLine` de la clase `Console`, estos métodos y propiedades no están precedidos por la palabra `static`. A tales métodos y propiedades se les denomina **métodos y propiedades de instancia** y tienen que ser invocados a través de una instancia de la clase:

```
Stopwatch crono = new Stopwatch();
crono.Restart();
//...
crono.Stop();
Console.WriteLine("Tiempo transcurrido {0} ms", crono.EllapsedMilliseconds);
```

Para crear una instancia de la clase hay que hacer

new <nombre de la clase>(<parámetros>)

Por ejemplo `new Stopwatch()`, porque en este caso no se necesitan parámetros.

Al aplicar el operador `new` se separa memoria para una instancia (objeto) del tipo definido por la clase y luego se aplica un **constructor** de la clase. La memoria separada para el objeto depende del tipo y cantidad de las variables de instancia de la clase. Un constructor es un método definido con el mismo nombre de la clase (línea 7 en el Listado 6.2). El constructor se encarga de inicializar la instancia con los valores necesarios.

Para ver un primer ejemplo de cómo definir nuestros propios tipos mediante clases vamos a definir una clase `MyStopwatch` (Listado 6.3) que nos da una implementación simplificada de la clase `Stopwatch`.

```
namespace WEBOO.Programacion
{
  public class MyStopwatch
  {
    public MyStopwatch();
    public bool IsRunning { get; }
    public long ElapsedMilliseconds { get; }
    public void Restart();
    public void Stop();
    public void Start();
  }
}
```

Listado 6.3 Versión simplificada de Stopwatch

6.2 Variables de instancia de una clase

Para la implementación de esta clase `MyStopwatch`, es necesaria alguna forma de medir el tiempo aún más elemental que la propia clase `Stopwatch`. Para ello se usará el método estático `TickCount` de la clase `Environment`, este método devuelve un valor entero que indica la cantidad de milisegundos transcurridos desde que se echó a andar el sistema. De modo que si censamos a `TickCount` cuando se haga `Restart` y luego volvemos a censar cuando se haga `Stop` o `EllapsedMilliseconds`, entonces la diferencia entre ambas mediciones nos dará aproximadamente el tiempo transcurrido.

Hace falta entonces que un `MyStopwatch` pueda "memorizar" el `TickCount` censado al empezar y el `TickCount` censado al parar. Esto se resolverá con las variables `arrancada` y `parada`. Además, se usará un tercera variable `andando` de tipo `bool` para indicar si está "caminando" (`true`) o "parado" (`false`) (Listado 6.4 líneas 3, 4 y 5).

Estas variables se denominan **variables de instancia** porque cada instancia creada con el operador `new` tiene su propio conjunto de tales variables (note que las variables no tienen la especificación `static`)

```
1   public class MyStopwatch
2   {
3     long arrancada;
4     long parada;
5     bool andando;

6     public MyStopwatch()
```

```
7     {
8       arrancada = 0;
9       parada = 0;
10      andando = false;
11    }
12    public bool IsRunning { get; }
13    public long ElapsedMilliseconds { get; }
14    public void Restart();
15    public void Stop();
16    public void Start();
17  }
```

Listado 6.4 Variables de Instancia y Constructor

De modo que al hacer

```
MyStopwatch crono = new MyStopwatch();
```

se reservaría memoria para la instancia y se pondría en `crono` una referencia a la misma (Figura 6.1)

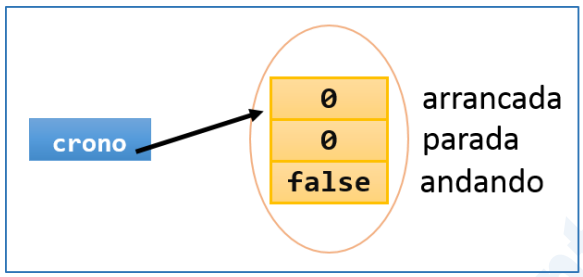

Figura 6.1 Representación de la memoria del objeto

Note que la variable `andando` es de tipo `bool` y en este caso servirá para indicar si el cronómetro está caminando (`true`) o si está detenido (`false`).

6.3 Constructores

Un **constructor** es un método que se aplica cuando se crea un objeto instancia de una clase al aplicar el operador `new`. Luego de separado el espacio para las variables de instancia, el constructor se encarga de inicializar los valores de esas variables, así como hacer todas las verificaciones y controles que sean necesarios para garantizar que quede un objeto en un estado "consistente" , es decir que cumpla con la semántica que se quiere tengan los objetos de la clase.

En C# un constructor es un método que tiene el mismo nombre de la clase. En el caso de `MyStopwatch`, el constructor lo que hace es inicializar las variables `arrancada`, `parada` y `andando` (Listado 6.4 líneas 6-11). De modo que en este caso un objeto `MyStopwatch` comienza en el estado detenido (o sea `andando` tiene valor `false`) con instante de arrancada `0` e instante de parada 0, si se le preguntara por el tiempo marcado la respuesta en este caso sería `0`.

En este ejemplo se ha incluido un constructor con un objetivo ilustrativo; realmente el operador `new` además de separar espacio para las variables de instancia inicializa éstas

con los valores predeterminados según el tipo de las mismas (en este caso `0` para `arrancada` y `parada` por ser numéricas y `false` para `andando` por ser `bool`). En este caso la definición del constructor podría haberse omitido.

Al igual que un método, un constructor puede tener parámetros. Es lógico que quien decida crear una instancia tenga que aportar información para la adecuada creación de la misma. Este es un principio básico que la POO promueve: que los objetos se creen en un estado consistente y con información coherente que luego los restantes métodos se deben encargar de mantener. La clase `Libro` (Listado 6.5) tiene un constructor que recibe dos parámetros `titulo` y `precio` para inicializar las variables de instancia que caracterizan a un objeto tipo `Libro`.

```
1  class Libro
2  {
3    string titulo;
4    float precio;
5    public Libro(string titulo, float precio)
6    {
7      this.titulo = titulo;
8      this.precio = precio;
9    }
10   public string Titulo
11   {
12     get { return titulo; }
13   }
14   public float Precio
15   {
16     get { return precio; }
17     set { precio = value; }
18   }
19   //...
20 }
```

Listado 6.5 Esbozo de una clase Libro

Al igual que una clase puede tener más de un método con el mismo nombre (siempre que se diferencien en el tipo o cantidad de los parámetros para que no haya ambigüedad), una clase puede tener más de un constructor.

6.3.1 La variable especial this

Note las líneas 7 y 8 del Listado 6.5 en donde se ha usado una especie de variable de nombre `this`. La palabra reservada `this`, utilizada dentro del código de un método o propiedad de una clase, se refiere al objeto a través del cual se ha invocado el método o propiedad. En este ejemplo, su utilización dentro del constructor se refiere al propio objeto que se está creando. Su utilización en este ejemplo ha servido para desambiguar si nos estamos refiriendo al parámetro de nombre `titulo` o la variable de instancia `titulo` del objeto, lo cual resuelve al escribir `this.titulo` (lo mismo aplica para el parámetro de nombre `precio` y la variable de instancia `precio`).

Esta variable especial `this` sirve también para cuando un método de la clase quiere invocar a un método de otra clase y pasarle como parámetro el objeto para el que está

ejecutando. Una instrucción como `x.F(this)` ejecutando dentro de un método `M` está llamando al método `F` de `x` y le está pasando como parámetro el propio objeto a través del cual se llamó a `M`.

Es una práctica de muchos desarrolladores utilizar la palabra reservada **`this`** para resaltar que se están refiriendo a las variables de instancia del objeto. Esto puede ser útil para aumentar la legibilidad del código, al distinguir las variables de instancia de los parámetros y variables locales de un método, aun cuando no haya ninguna situación que desambiguar.

6.4 Propiedades

Una propiedad devuelve un valor con información sobre el objeto. En el ejemplo de `MyStopwatch` (lineas 7-18 del Listado 6.6) `IsRunning` es un propiedad que devuelve un valor `bool` para decir si el cronómetro está andando o está detenido, y `ElapsepMilliseconds` es una propiedad que devuelve el tiempo transcurrido hasta el momento en que se paró, si el cronómetro está detenido, o hasta el momento en que se está consultando la propiedad si el cronómetro está caminando.

```
1  public class MyStopwatch
2  {
3    long arrancada;
4    long parada;
5    bool andando;
6    public Stopwatch();
7    public bool IsRunning
8    {
9      get { return andando; }
10   }
11   public long ElapsedMilliseconds
12   {
13     get
14     {
15       if (IsRunning) return Environment.TickCount - arrancada;
16       else  return parada - arrancada;
17     }
18   }
19   public void Restart();
20   public void Stop();
21   public void Start();
22 }
```

Listado 6.6 Propiedades

6.4.1 Parte get de una propiedad

La sección `get` dentro de la definición de la propiedad describe el código que se ejecutará cuando la propiedad sea consultada a través de un objeto.

De modo que cuando se ejecute un código como `Console.WriteLine(crono.IsRunning)` se ejecutará el código `return andando` (línea 9 Listado 6.6).

El código de la parte `get` de una propiedad es como el código de un método, es decir debe garantizarse que siempre se salga de éste a través de un `return <expresión>` donde la expresión debe ser del tipo indicado para la propiedad.

No siempre el valor que devuelve una propiedad es un valor directamente almacenado en una variable de instancia. Por ejemplo, en este ejemplo no interesa mostrar las variables `arrancada` y `parada`, sino que la propiedad `ElapsedMilliseconds` calcula y devuelve el valor del tiempo transcurrido según si el cronómetro está caminando o no (Listado 6.6 líneas 15 y 16).

6.4.2 Parte set de una propiedad

En este ejemplo de `MyStopwatch` no tiene sentido que el código cliente quiera por su cuenta cambiar el valor de las propiedades `IsRunning` o `ElapsedMilliseconds`, sino que estas cambian según se apliquen otros métodos de la clase (`Start`, `Restart` y `Stop` en este caso). Sin embargo, una propiedad puede tener también una parte `set`, y eso quiere decir que el nombre de la propiedad se puede usar en la parte derecha de una asignación. El código de la clase `Libro` (Listado 6.5) tiene una propiedad `Titulo` con parte `get` y una propiedad `Precio` que tiene parte `get` y parte `set`.

La propiedad `Titulo` no tiene parte `set` porque en este caso una vez creado un objeto `Libro` no se necesita cambiar el título. Sin embargo, la propiedad `Precio` si tiene parte `set` porque el precio del libro pudiera variar.

La palabra reservada `value` utilizada dentro de la implementación de la parte `set` se refiere al valor de la parte derecha de la asignación en que se ha utilizado la propiedad y que ha dado lugar a la ejecución de la parte `set`. Si se hace `miLibro.Precio = 20;` al ejecutar la parte `set` la variable especial `value` tendrá en este caso el valor `20`.

La parte `set` puede usarse para hacer algo más que asignar directamente un valor a una variable de instancia, por ejemplo para aplicar un descuento al precio si se cumple determinada condición (Listado 6.7)

```
public float Precio
{
  get { return precio; }
  set
  {
    if (condicion) precio = value - descuento;
    else precio = value;
  }
}
```

Listado 6.7 Aplicando descuento al precio

6.4.3 Implementación Implícita de Propiedades

La clase `Libro` del Listado 6.5 podría haberse definido de modo más simple como se muestra en Listado 6.8. Note que en este caso no se han definido variables de instancias, el compilador transforma esta definición en una equivalente a la del Listado 6.5 y nos ahorra de tener que estar definiendo dichas variables de instancia.

```
class Libro
{
  public Libro( string titulo, float precio )
  {
    Titulo = titulo;
    Precio = precio;
  }
  public string Titulo { get; private set; }
  public float Precio { get; set; }
  //...
}
```

Listado 6.8 Propiedades con implementación implícita

Note que no se ha escrito una implementación de la propiedad `Titulo`, solo se ha indicado que ésta es `{ get; private set; }`. La indicación `private` al `set` significa que esta propiedad es privada para los clientes de la clase y que solo puede ser modificada por los métodos de la clase, como es el caso en este ejemplo en que es utilizada para asignarle un valor en el constructor.

Las variables de instancia son necesarias para la implementación de la clase porque constituyen "la memoria" que expresa "el estado" del objeto, pero puede que no sean de interés para el código cliente (como fue el caso de las variables de instancia `arrancada` y `parada` de la clase `MyStopwatch`). No es común tener variables de instancia que sean públicas y que por tanto puedan ser consultadas y modificadas por el código cliente. Si usted quiere lograr el efecto de permitir consultar una variable de instancia pero que no se pueda modificar (como es el caso de la variable `andando` en la clase `MyStopwatch`), utilice entonces una propiedad definida como `{get; private set;}`. Aún en el poco frecuente caso en que quiera que una variable de instancia pueda ser consultada y modificada, utilice para ello una propiedad pública y que tenga `{get; set;}`, lo que deja el camino abierto para modificar la implementación del código (por ejemplo, cambiar la implementación de `Libro` a una que pueda hacerle descuentos al precio) sin que esto implique tener que hacer cambios en el código cliente de la clase (el código en el que se está usando la clase).

6.5 Constantes simbólicas y readonly

6.5.1 Modificador const

Una expresión constante es una expresión que puede ser evaluada en tiempo de compilación y que permite por tanto especificar valores constantes que no variarán durante la ejecución.

Las constantes son únicas para todas las instancias de la clase y desde fuera de la clase se acceden a través del nombre de la clase. Es decir, que se utilizan igual que los miembros especificados `static`. Por ejemplo, la clase `MyMath` (Listado 6.9) incluye dos constantes `pi` y `e`.

```
class MyMath
```

```
{
  public const double pi = 3.14159;
  public const double e = 2.71828;
  //...
}
```

Listado 6.9 Definición de constantes en una clase MyMath

Entonces estas constantes se pueden usar desde fuera de la clase `MyMath` escribiendo `MyMath.pi` y `MyMath.e`.

Realmente .NET en su biblioteca BCL ofrece la clase `System.Math` que tiene muchas utilidades matemáticas.

El tipo especificado en la declaración de la constante puede ser cualquiera de los tipos primitivos numéricos, `char`, `bool`, `string`, o un tipo `enum`. Una constante puede participar dentro de otras expresiones, incluyendo la definición de otras constantes o de miembros de una declaración de un tipo enumerativo. Vea la definición de la clase `MyMath` a la que se le ha añadido una nueva constante `GradoEnRadianes`, que es una expresión en la que participa la constante `pi`.

```
class MyMath
{
  public const double pi = 3.14159;
  public const double e = 2.71828;
  public const double GradoEnRadianes = pi / 180;
  public const int MaxLong = 100;
  //...
}
```

Listado 6.10 Expresiones constantes

Esta constante `GradoEnRadianes` a su vez nos permitiría convertir ángulos de grados a radianes de la siguiente manera:

```
int angGrados = 60;
double angRadianes = angGrados * MyMath.GradoEnRadianes;
```

En el caso en que haya constantes que dependan unas de otras, el compilador C# de manera automática las evaluará en el orden apropiado, siempre y cuando no provoquen una circularidad, en cuyo caso dará error en tiempo de compilación. Este problema se presenta cuando el cómputo del valor de una variable depende de otra u otras que de manera directa o indirecta también dependen de la variable inicial. Esto comúnmente se conoce como el problema de la referencia circular. ¡Si quiere visualizarlo, imagínese un perro que se intenta morder su propia cola!.

6.5.2 Modificador readonly

El modificador `const` sólo es utilizable para definir constantes en aquellos tipos para los que existe una notación literal para denotar sus valores, como es el caso de los tipos primitivos numéricos, `char`, `bool` y los tipos `string`, o los definidos por `enum`.

¿Cómo ponerle la especificación `const` a una variable `Origen` que es de un tipo `Punto` si no se tiene una notación para expresar valores literales de puntos? Habría que poder hacer algo así como

```
const Punto Origen = new Punto(0,0);
```

Pero en tiempo de compilación no se puede aplicar el operador `new` y el constructor de `Punto`. ¿Cómo lograr entonces que en tiempo de ejecución se pueda disponer de una entidad `Origen` que represente a un objeto de tipo `Punto` con las coordenadas `0,0` y que no se pueda cambiar el valor de `Origen`?

Una variable de instancia, o una variable estática, puede estar precedida por el modificador `readonly`. De este modo, se pudiera haber definido dentro de la clase `Punto` (Listado 6.11) lo siguiente:

```
class Punto{
  public static readonly Punto Origen;
  public int X{ get; private set; }
  public int Y{ get; private set; }
  public Punto(int x, int y)
  {
    X = x; Y = y;
  }
  public double Distancia( Punto p )
  {
    return Math.Sqrt((p.X - X) * (p.X - X) +
                     (p.Y - Y) * (p.Y - Y));
  }
  //Otros métodos y propiedades de Punto
}
```

Listado 6.11 Definición de una clase Punto

Esto significa entonces que se puede usar `Punto.Origen` para trabajar con el objeto `Punto` que indica el origen de coordenadas. Por ejemplo

```
p.Distancia(Punto.Origen)
```

calcula la distancia del punto `p` al origen.

Que `Origen` se haya especificado `readonly` significa que un intento de hacer

```
Punto.Origen = new Punto(100, 200);
```

será reportado como error por el compilador.

Es decir, añadirle el modificador `readonly` (sólo lectura) a una variable significa que la tal "variable" no lo será tanto porque no podrá asignársele un nuevo valor durante la ejecución.

6.5.3 Dándole valor a una variable readonly

¿Dónde asignarle el valor original a una variable `readonly` para que luego permanezca constante? La asignación a una variable `readonly` solo se puede hacer en un constructor. Si la variable es `readonly` y a su vez `static` la inicialización de la variable se hace en lo que se denomina un **inicializador estático**. Un inicializador estático es un método con el mismo nombre de la clase al que le ha añadido la especificación `static` y el cual se ejecuta solo una vez al iniciar la aplicación que utilice dicha clase. El Listado 6.12 nos muestra cómo se ha incluido un tal inicializador a la clase `Punto` con el propósito de inicializar la variable `Origen`.

```
class Punto{
  public static readonly Punto Origen;
  public int X{ get; private set; }
  public int Y{ get; private set; }
  public Punto(int x, int y)
  {
    X = x; Y = y;
  }
  static Punto()
  {
    Origen = new Punto(0,0);
  }
  //Métodos y propiedades de Punto
}
```

Listado 6.12 Inicializador estático para la clase Punto

A una variable de instancia que sea `readonly` solo se le puede asignar un valor dentro de un constructor. Es decir, que una vez creado un objeto dicha variable no podrá cambiar de valor.

La diferencia entre usar una variable de instancia `readonly` o una propiedad que sea `private set` es que con la variable `readonly` el código cliente tiene la garantía de que una vez creado el objeto el valor de dicha variable no variará, mientras que en el caso de la propiedad el ser `private set` le impide al código cliente cambiar el valor de la propiedad pero dicho valor podrá variar, porque cualquier método de la clase (y no solo el constructor) lo podrá cambiar. Cuándo usar una forma u otra es un asunto de estilo y diseño. El ejemplo de la sección siguiente nos ilustra los diferentes escenarios.

6.6 Ejemplo de una clase Cuenta

La clase `Cuenta` (Listado 6.13) nos ilustra los diferentes escenarios estudiados anteriormente.

```
public class Cuenta
{
  public static readonly double SaldoMinimo = 1000;
  public readonly string Titular;
  public double Saldo { get; private set; }
  public Cuenta(string titular, double saldoInicial) {
```

```
    Titular = titular;
    if (saldoInicial < SaldoMinimo)
      throw new Exception("Hay que abrir una cuenta con un saldo mínimo");
    Saldo = saldoInicial;
  }
  public void Deposita(double cantidad) {
    if (cantidad <= 0)
      throw new Exception("Cantidad a depositar debe ser mayor que cero");
    Saldo += cantidad;
  }
  public void Extrae(float cantidad) {
    if (cantidad <= 0)
      throw new Exception("Cantidad a extraer debe ser mayor que cero");
    else if (Saldo - cantidad < SaldoMinimo)
      throw new Exception("No hay suficiente saldo para extraer");
    Saldo -= cantidad;
  }
}
```

Listado 6.13 Una clase Cuenta

La variable `SaldoMinimo` es `static` porque es un mismo valor para todas las instancias de la clase (todas las cuentas deben respetar el mismo saldo mínimo) y es `readonly` porque dicho valor no cambiará durante toda la ejecución.

La variable `Titular` no es `static` porque es una variable de instancia, ya que cada cuenta tiene un titular, pero es `readonly` porque una vez creada una cuenta no se podrá cambiar el titular.

La propiedad `Saldo` tiene `get` público porque un código cliente podrá consultar el saldo de la cuenta. Sin embargo, tiene `private set` porque un código cliente no podrá cambiar directamente el saldo de una cuenta, pero sí podrá lograrlo a través de la aplicación de los métodos `Deposita` y `Extrae`.