Propiedades

Una propiedad es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Se pueden usar las propiedades como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados *descriptores de acceso*. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

En este ejemplo, la clase **PeriodoDeTiempo** almacena un período de tiempo. Internamente, la clase almacena el tiempo en segundos, pero una propiedad denominada **Horas** permite especificar el tiempo en horas. Los descriptores de acceso para la propiedad **Horas** realizan la conversión entre horas y segundos.

```
class PeriodoDeTiempo
{
   private double segundos;

   public double Horas
   {
      get { return segundos / 3600; }
      set { segundos = value * 3600; }
   }
}


class Program
{
   static void Main()
   {
      PeriodoDeTiempo t = new PeriodoDeTiempo();

      // La asignación de la propiedad Horas provoca la llamada del descriptor de
acceso 'set'.
      t.Horas = 24;

      // La evaluación de la propiedad Horas provoca la llamada del descriptor de
acceso 'get'.
      System.Console.WriteLine("Tiempo en horas: " + t.Horas);
```

```
    }
}
```

Definiciones de cuerpos de expresión

Es habitual tener propiedades que simplemente devuelvan de inmediato el resultado de una expresión. Hay un acceso directo de sintaxis para definir estas propiedades mediante =>, se lo llama operador lambda:

```
public string NombreCompleto => Nombre + " " + Apellido;
```

La propiedad debe ser de solo lectura, y no se utiliza la palabra clave de descriptor de acceso get.

Propiedades autoimplementadas

En C# 3.0 y versiones posteriores, las propiedades implementadas automáticamente hacen que la declaración de propiedades sea más concisa cuando no es necesaria ninguna lógica adicional en los descriptores de acceso de la propiedad. También permite que el código cree objetos. Cuando se declara una propiedad tal como se muestra en el ejemplo siguiente, el compilador crea un campo de respaldo privado y anónimo al que solo se puede acceder con los descriptores de acceso de propiedad get y set.

En el ejemplo siguiente se muestra una clase simple que tiene algunas propiedades implementadas automáticamente:

```
class Cliente
{
        // Propiedades autoimplementadas
        public double TotalCompras { get; set; }
        public string Nombre { get;  set; }
        public int ClienteID { get; set; }
        // Constructor
```

```
        public Cliente(double compras, string nombre, int ID)
        {
            TotalCompras = compras;
            Nombre = nombre;
            ClienteID = ID;
        }

        // Métodos
        public string RecuperarInfoDeContacto() { return "Información de contacto del
cliente"; }
        public  string   RecueprarHistorialDeTransacciones() {     return    "Historial de
compras del cliente"; }
        // .. Más métodos, eventos, etc., de la clase Cliente.
}

class Program
{
        static void Main()
{
            // Creación de un nuevo objeto cliente, llamando al constructor sobrecargado.
            Cliente clie = new Cliente(4987.63, "El Trevol", 90108);

            // Modificación de la propiedad autoimplementada TotalCompras
clie.TotalCompras += 499.99;
        }
 }
```

En C# 6 y versiones posteriores, puede inicializar las propiedades implementadas automáticamente de forma similar a los campos:

```
public  string Nombre { get;  set; } = "María";
```

Información general sobre propiedades

- Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, a la vez que se oculta el código de implementación o comprobación.

- Se utiliza un descriptor de acceso de propiedad **get** para retornar el valor de propiedad y un descriptor **set** para asignarle un nuevo valor a la propiedad. Estos descriptores de acceso pueden tener diferentes niveles de acceso, es decir distinta visibilidad.
- La palabra clave **value** se utiliza para definir el valor asignado por el descriptor de acceso **set**. En definitiva el valor que el usuario ingrese en dicho campo llega como parámetro a la propiedad con la palabra clave **value**.
- Las propiedades que no implementan un descriptor de acceso **set** son de solo lectura. También pueden tener un descriptor **set** privado y de esta manera ser de solo lectura. Otra manera de emular este comportamiento es declarar el miembro con la visibilidad **readonly**, de esta manera los valores de la variable no podrán cambiar en ningún hilo de ejecución (thread) de la aplicación.
- Cuando se trate de propiedades sencillas que no requieran ningún código de descriptor de acceso personalizado, considere la posibilidad de utilizar propiedades implementadas automáticamente.

Métodos, firmas de método y sobrecargas

Los métodos definen la funcionalidad de la clase, y se definen como una función de C#. Se declaran en una clase o struct especificando el nivel de acceso, como public o private, y modificadores opcionales como abstract o sealed, el valor de retorno, el nombre del método y cualquier parámetro de método. Todas estas partes de la declaración del método forman la **firma del método**.

En el siguiente ejemplo el método se llama MostrarNombreCompleto, y su firma está formada el nivel de acceso **public**, el valor de retorno **string**, y los dos parámetros **nombre** y **apellido**, ambos tipo **string**:

```
public string MostrarNombreCompleto(string nombre, string apellido)
{
   return nombre+' ' +apellido;
}
```

Esta firma puede ser diferente en sus tipos, dado que no se reconoce el nombre de cada variable como parte de la firma sino más bien los tipos que recibe. Pueden existir entonces

infinitos Métodos o funciones con el mismo nombre siempre y cuando la cantidad y orden de los tipos de dato que reciba sean diferentes.

Sobrecargando el método anterior para que reciba la inicial del segundo nombre, cambia la firma a el nivel de acceso **public**, el valor de retorno **string**, y los tres parámetros **nombre, otroNombre** y **apellido**, tipos **string, char, string**:

```
public string MostrarNombreCompleto(string nombre, char otroNombre, string apellido)
{
   return nombre + ' ' + otroNombre + ' ' + apellido;
}
```

No obstante para la sobrecarga se necesita una combinación de nombre de clase con parámetros diferentes. Mismo nombre de clase con mismos parámetros no compila aunque el valor de retorno sea diferente. Aunque sea parte de la firma.
Los parámetros de un método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros. Esta clase contiene tres métodos con código propio definido y uno abstracto a ser implementado por la clase que herede de la clase Moto:

```
  abstract class Moto
   {
      // Este método se puede llamar de cualquier lugar porque es public.
      public void IniciarMotor() {/* Aquí iría el código */ }

      // Solo clases derivadas pueden ver este método porque es protected
      protected void CargarNafta(int litros) {/* Aquí iría el código */ }

      // Clases derivadas pueden sobrescribir el código de la clase base porque es virtual.
public virtual int Conducir(int kilómetros, int velocidad)  {/* Aquí iría el código */return 1; }

      // Las clases derivadas pueden implementarlo.
      public abstract double ObtenerVelMáxima();
   }
```

Constructores

Cada vez que se crea una clase o struct, se llama a su constructor. El constructor es un método que se llama cuando se crea la clase o struct. Una clase o struct puede tener varios constructores que toman argumentos diferentes, esto se denomina sobrecarga de constructores. Los constructores permiten al programador establecer valores predeterminados, limitar la creación de instancias y escribir código flexible y fácil de leer. Si no proporciona un constructor para el objeto, C# creará uno de forma predeterminada que cree instancias del objeto y establezca las variables miembro en los valores predeterminados.

Constructores de Instancia

Los constructores de instancias se usan para crear e inicializar cualquier variable de miembro de instancia cuando se usa la expresión **new** para crear un objeto de una clase.

En el ejemplo vemos la clase **Coordenadas** con el constructor por omisión, en el cual se le agregó la inicialización de las variables x e y:

```
class Coordenadas
{
        public int x, y;

        // constructor  public Coordenadas ()
        {       x = 0;
                y = 0;
        }
}
```

Constructor con dos argumentos

Aquí sobrecargamos el constructor del ejemplo anterior, teniendo otro que recibe dos argumentos.

```
public Coordenadas (int x, int y)
        {
        this.x = x;
        this.y = y;
        }
```

Esto permite crear objetos Coordenadas con valores iniciales concretos o predeterminados, del siguiente modo:

```
Coordenadas p1 = new Coordenadas ();
Coordenadas p2 = new Coordenadas (5, 3);
```

Si una clase no tiene un constructor, se genera automáticamente un constructor predeterminado y se usan valores predeterminados para inicializar los campos de objeto. Por ejemplo, un int se inicializa a 0. Por consiguiente, como el constructor predeterminado de la clase Coordenadas inicializa todos los miembros de datos en cero, se puede quitar del todo sin cambiar el funcionamiento de la clase. Los constructores de instancia también se pueden utilizar para llamar a los constructores de instancia de clases base. El constructor de clase puede invocar el constructor de la clase base a través del inicializador, en el siguiente ejemplo, el constructor de la clase Circulo que hereda de la clase Figura llama al constructor de su clase base que es Figura:

```
class Circulo : Figura
{
    public Circulo (double radio) : base(radio, 0)
    {
    }
}
```

ToString()

Object.ToString es el principal formato de método en .NET Framework. Convierte un objeto en su representación de cadena de caracteres así será apropiado para su presentación. En las implementaciones donde no se tenga un método **ToString** de forma predeterminada el

método **Object.ToString** devuelve el nombre completo incluyendo el namespace del tipo de objeto.