برنامه نویسی پیشرفته
C#

۲۶ آبان ۹۸

ملکی مجد

Implementing **properties** to access fields

motivation

- Consider the following structure that represents a position on a computer screen as a pair of coordinates, x and y. Assume that the range of valid values for the x-coordinate lies between 0 and 1280, and the range of valid values for the y-coordinate lies between 0 and 1024.

motivation (1)

```
struct ScreenPosition
{
    public int X;
    public int Y;

    public ScreenPosition(int x, int y)
   {
            this.X = rangeCheckedX(x);
            this.Y = rangeCheckedY(y);
    }
}
```

```
private static int rangeCheckedX(int x) {
       if (x < 0 || x > 1280) {
       throw new ArgumentOutOfRangeException("X");
       }
        return x;
}

private static int rangeCheckedY(int y) {
       if (y < 0 || y > 1024) {
      throw new ArgumentOutOfRangeException("Y");
      }
      return y;
}
```

Problem with ScreenPosition?

- Public data is often a bad idea because the class cannot control the values that an application specifies

- Examplew - the *ScreenPosition* constructor checks but …

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // oops
```

Solution to the access problem

- The common way to solve this problem
  - make the fields private and add an accessor method and a modifier method to respectively read and write the value of each private field.

```
struct ScreenPosition{
        ...
        public int GetX()
                { return this.x; }
        public void SetX(int newX)
                { this.x = rangeCheckedX(newX); }
        ...
        private static int rangeCheckedX(int x) { ... }
        private static int rangeCheckedY(int y) { ... }
        private int x, y;
}
```

The price of the proposed solution

- *ScreenPosition* no longer has a natural field-like syntax
- it uses awkward method-based syntax instead.

```
~~origin.X += 10;~~
int xpos = origin.GetX();
origin.SetX(xpos + 10);
```

Are you motivated to use properties?

- There is no doubt that, in this case, **using public fields is syntactically cleaner, shorter, and easier.** Unfortunately, using public fields breaks encapsulation. By using **properties**, you can combine the best of **both** worlds (fields and methods) to retain encapsulation while providing a field-like syntax.

What are properties?

- A *property* is a cross between a field and a method
  - it looks like a field
  - acts like a method

- The syntax for a property declaration

```
AccessModifier Type PropertyName
{
        get { // read accessor code }
        set { // write accessor code }
}
```

ScreenPosition with property

```
struct ScreenPosition{
    private int _x, _y;
    public ScreenPosition(int X, int Y) {
        this._x = rangeCheckedX(X);
        this._y = rangeCheckedY(Y);
    }
    private static int rangeCheckedX(int x){ ... }
    private static int rangeCheckedY(int y) { ... }
}
```

```
public int X {
    get { return this._x; }
    set {this._x = rangeCheckedX(value);}
}
public int Y {
    get { return this._y; }
    set {this._y = rangeCheckedY(value);}
}
```

ScreenPosition with property

```
struct ScreenPosition{
     private int _x, _y;
     public ScreenPosition(int X, int Y) {
           this._x = rangeCheckedX(X);    this._y =
rangeCheckedY(Y);
     }
     private static int rangeCheckedX(int x){ ... }
     private static int rangeCheckedY(int y) { ... }
}
```

```
public int X {
     get { return this._x; }
     set {this._x = rangeCheckedX(value);}
}
public int Y {
     get { return this._y; }
     set {this._y = rangeCheckedY(value);}
}
```

Lowercase *_x* and *_y* are **private fields**.
Uppercase *X* and *Y* are **public properties**.
All *set* accessors are passed the data to be written by using a hidden, built-in parameter named ***value***.

Note

- In this example, a private field directly implements each property, but this is only one way to implement a property.
- All that is required is for a *get* accessor to return a value of the specified type. Such a value can easily be **calculated dynamically** rather than being simply retrieved from stored data, in which case there would **be no need for a physical field**.
- The definition of properties are equally applicable to classes; the syntax is the same.

Using properties

- When you use a property in an expression, you can use it in a read context (when you are retrieving its value) and in a write context (when you are modifying its value).

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X; // calls origin.X.get()
int ypos = origin.Y; // calls origin.Y.get()

origin.X = 40; // calls origin.X.set, with value set to 40
origin.Y = 100; // calls origin.Y.Set, with value set to 100
```

Read-only properties

```
struct ScreenPosition
{
        private int _x;
        public int X
        {
            get { return this._x; }
        }
}

origin.X = 140; // compile-time error
```

Write-only properties

```
struct ScreenPosition
{
        private int _x;
        ...
        public int X
        {
                set { this._x = rangeCheckedX(value);
        }
}

Console.WriteLine(origin.X); // compile-time error
origin.X = 200; // compiles OK
origin.X += 10; // compile-time error
```

Property accessibility

it is possible within the property declaration to override the property accessibility for the *get* and *set* accessors

```
struct ScreenPosition
{
        private int _x, _y;
        ...
        public int X {
                get { return this._x; } //public
                private set { this._x = rangeCheckedX(value); }
        }
        public int Y {
                get { return this._y; }
                private set { this._y = rangeCheckedY(value); }
        }
         ...
}
```

Property accessibility

- You can change the accessibility of only one of the accessors when you define it.
- The modifier must not specify an accessibility that is less restrictive than that of the property

```
struct ScreenPosition
{
        private int _x, _y;
        ...
        public int X {
                get { return this._x; } //public
                private set { this._x = rangeCheckedX(value); }
        }
        public int Y {
                get { return this._y; }
                private set { this._y = rangeCheckedY(value); }
        }
         ...
}
```

property restrictions

- You can assign a value through a property of a structure or class only after the structure or class has been initialized
- You can't use a property as a *ref* or an *out* argument to a method (although you can use a writable field as a *ref* or an *out* argument).
- A property can contain at most one *get* accessor and one *set* accessor. A property cannot contain other methods, fields, or properties.
- The *get* and *set* accessors cannot take any parameters. The data being assigned is passed to the *set* accessor automatically by using the *value* variable.
- You can't declare properties by using *const*,

Declaring interface properties

Interfaces can define properties as well as methods.

```
interface IScreenPosition
{
        int X { get; set; }
        int Y { get; set; }
}
```

- Any class or structure that implements this interface must implement the *X* and *Y* properties with *get* and *set* accessor methods.

EXAMPLE

```
struct ScreenPosition : IScreenPosition{
    public int X
   {
            get { ... }
            set { ... }
    }
    public int Y
    {
            get { ... }
            set { ... }
     }
}
```

declare the property implementations as virtual

```
class  ScreenPosition : IScreenPosition{
    public virtual int X
   {
              get { ... }
              set { ... }
    }
    public virtual  int Y
    {
              get { ... }
              set { ... }
     }
}
```

implement a property by using the explicit interface implementation

- An explicit implementation of a property is **nonpublic** and **nonvirtual** (and cannot be overridden).

```
struct ScreenPosition : IScreenPosition{
   int IScreenPosition.X {
           get { ... }
           set { ... }
   }
   int IScreenPosition.Y  {
           get { ... }
           set { ... }
    }
}
```

Simple get and set

- The principal purpose of properties is to hide the implementation of fields from the outside world
- The value of the *get* and *set* accessors of **simply wrap operations** :
  - **Compatibility with applications**
  - **Compatibility with interfaces**

Generating automatic properties

```
class Circle
{
        public int Radius{ get; set; }
        ...
}
```

```
class Circle{
        private int _radius;
        public int Radius{
                get { return this._radius;}
                set { this._radius = value;}
        }
        ...
}
```

C# compiler converts

note

- The **syntax** for defining an **automatic property** is almost identical to the syntax for defining a **property in an interface**. The exception is that an **automatic property can specify an access modifier** such as *private*, *public*, or *protected*

a read-only automatic property

```
class Circle
{
        public DateTime CircleCreatedDate { get; }
}
```

This is **useful** in scenarios where you want to create an **immutable property**; a property that is set when the object is constructed and cannot subsequently be changed.

the date on which an object was created, the name of the user who created it, generate a unique identifier value

a read-only automatic property - initialize

```
class Circle{
     public Circle() {
         CircleCreatedDate = DateTime.Now;
    }
    public DateTime CircleCreatedDate { get; }
}
```

or

```
class Circle{
            public DateTime CircleCreatedDate { get; } = DateTime.Now;
}
```

Initializing objects by using properties

```
public class Triangle
{
        private int side1Length;
        private int side2Length;
        private int side3Length;
        public Triangle(int length1, int length2, int length3)
        {
                this.side1Length = length1;
                this.side2Length = length2;
                this.side3Length = length3;
        }
}
```

What if the various combinations you want to enable for initializing the fields?

Initializing objects by using properties

```
public class Triangle
{
        private int side1Length = 10;
        private int side2Length = 10;
        private int side3Length = 10;
        public int Side1Length{set { this.side1Length = value; }}
        public int Side2Length { set { this.side2Length = value; } }
        public int Side3Length { set { this.side3Length = value; } }
}
```

Initializing objects by using properties

```
Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12,Side3Length = 15 };

Triangle tri5 = new Triangle("Equilateral triangle") { Side1Length = 3, Side2Length = 3, Side3Length = 3 };
```

The important point to remember is that the constructor runs first and the properties are set afterward.

Write code

- Write class polygon
- Fields:
  - **NumSides (int)**
  - **SideLength (double)**
- Use property for fileds
- Constructor
  - By default 4 and 10.0
- New three objects
  - Default (square)
  - Tree sided (triangle)
  - 5sided and 15.5 (polygon)
- Write the objects!