CHAPTER 15

Properties

Properties in C# provide the ability to protect a field by reading and writing to it through special methods called *accessors*. They are generally declared as `public` with the same data type as the field they are going to protect, followed by the name of the property and a code block that defines the get and set accessors.

```
class Time
{
  private int seconds;

  public int sec
  {
    get { return seconds; }
    set { seconds = value; }
  }
}
```

Properties are implemented as methods, but used as though they are fields.

```
static void Main()
{
  Time t = new Time();
  int s = t.sec;
}
```

Note that the contextual `value` keyword corresponds to the value assigned to the property.

Auto-implemented properties

The kind of property where the get and set accessors directly correspond to a field is very common. Because of this there is a shorthand way of writing such a property, by leaving out the accessor code blocks and the private field. This syntax was introduced in C# 3.0 and is called an auto-implemented property.

```
class Time
{
  public int sec
  {
    get;
    set;
  }
}
```

Property advantages

Since there is no special logic in the property above, it is functionally the same as if it had been a public field. However, as a general rule public fields should never be used in real world programming because of the many advantages that properties bring.

First of all, properties allow a programmer to change the internal implementation of the property without breaking any programs that are using it. This is of particular importance for published classes, which may be in use by other programmers. In the `Time` class for example, the field's data type could need to be changed from `int` to `byte`. With properties, this conversion could be handled in the background. With a public field, however, changing the underlying data type for a published class will likely break any programs that are using the class.

```
class Time
{
  private byte seconds;

  public int sec
  {
    get
    {
      return (int)seconds;
    }
    set
    {
      seconds = (byte)value;
    }
  }
}
```

A second advantage of properties is that they allow the programmer to validate the data before allowing a change. For example, the `seconds` field can be prevented from being assigned a negative value.

```
set
{
  if (value > 0)
    seconds = value;
  else
    seconds = 0;
}
```

Properties do not have to correspond to an actual field. They can just as well compute their own values. The data could even come from outside the class, such as from a database. There is also nothing that prevents the programmer from doing other things in the accessors, such as keeping an update counter.

```
public int hour
{
  get
  {
    return seconds / 3600;
  }
  set
  {
    seconds = value * 3600;
    count++;
  }
}

private int count = 0;
```

Read-only and write-only properties

Either one of the accessors can be left out. Without the set accessor the property becomes read-only, and by leaving out the get accessor instead the property is made write-only.

```
// Read-only property
private int sec
{
  public get { return seconds; }
}

// Write-only property
private int sec
{
  public set { seconds = value; }
}
```

Property access levels

The accessor's access levels can be restricted. For instance, by making the set property private.

```
private set { seconds = value; }
```

The access level of the property itself can also be changed to restrict both accessors. By default, the accessors are public and the property itself is private.

```
private int sec { get; set; }
```