6a

Classes and Objects: A Deeper Look

OBJECTIVES

In this lecture you will learn:

- Encapsulation and data hiding.
- The concepts of data abstraction and abstract data types (ADTs).
- To use keyword `this`.
- To use indexers to access members of a class.
- To use `static` variables and methods.
- To use `readonly` fields.
- To take advantage of C#'s memory-management features.

OBJECTIVES

- How to create a class library.
- When to use the `internal` access modifier.
- To use object initializers to set property values as you create a new object.
- To add functionality to existing classes with extension methods.
- To use delegates and lambda expressions to pass methods to other methods for execution at a later time.
- To create objects of anonymous types.

Outline

Outline

Time1 Class Declaration

- Class `Time1` (Fig. 10.1) represents the time of day.

Time1.cs

(1 of 2)

```
1  // Fig. 10.1: Time1.cs
2  // Time1 class declaration maintains the time in 24-hour format.
3  public class Time1
4  {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // set a new time value using universal time; ensure that
10    // the data remains consistent by setting invalid values to zero
11    public void SetTime( int h, int m, int s )
12    {
13       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
14       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
15       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
16    } // end method SetTime
17
```

Ensure that time values are within the acceptable range for universal time.

Fig. 10.1 | Time1 class declaration maintains the time in 24-hour format. (Part 1 of 2.)

Time1.cs

(2 of 2)

```
18    // convert to string in universal-time format (HH:MM:SS)
19    public string ToUniversalString()
20    {
21       return string.Format( "{0:D2}:{1:D2}:{2:D2}",
22          hour, minute, second );
23    } // end method ToUniversalString
24
25    // convert to string in standard-time format (H:MM:SS AM or PM)
26    public override string ToString()
27    {
28       return string.Format( "{0}:{1:D2}:{2:D2} {3}",
29          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30          minute, second, ( hour < 12 ? "AM" : "PM" ) );
31    } // end method ToString
32 } // end class Time1
```

Use `static` method `Format` of class `string` to return a `string` containing the formatted `hour`, `minute` and `second` values, each with two digits and, a leading `0` if needed.

To enable objects to be implicitly converted to their string representations, we need to declare method `ToString` with keyword `override`.

Fig. 10.1 | Time1 class declaration maintains the time in 24-hour format. (Part 2 of 2.)

10.2 `Time` Class Case Study (Cont.)

- A class's public methods are the **`public` services** or the **`public` interface** that the class provides to its clients.
- When instance variables are declared in the class body, they can be initialized using the same initialization syntax as a local variable.

Software Engineering Observation 10.1

Methods and properties that modify the values of `private` variables should verify that the intended new values are valid. If they are not, they should place the `private` variables in an appropriate consistent state.

- `string`'s `static` method `Format` is similar to the `string` formatting in method `Console.Write`, except that `Format` returns a formatted `string` rather than displaying it in a console window.

- The `Time1Test` application class (Fig. 10.2) uses class `Time1`.

`Time1Test.cs`

(1 of 2)

```
1  // Fig. 10.2: Time1Test.cs
2  // Time1 object used in an application.
3  using System;
4
5  public class Time1Test
6  {
7     public static void Main( string[] args )
8     {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // invokes Time1 constructor
11
12       // output string representations of the time
13       Console.Write( "The initial universal time is: " );
14       Console.WriteLine( time.ToUniversalString() );
15       Console.Write( "The initial standard time is: " );
16       Console.WriteLine( time.ToString() );
17       Console.WriteLine(); // output a blank line
18
19       // change time and output updated time
20       time.SetTime( 13, 27, 6 );
```

`new` invokes class `Time1`'s default constructor, since `Time1` does not declare any constructors.

Fig. 10.2 | `Time1` object used in an application. (Part 1 of 2.)

```
21       Console.Write( "Universal time after SetTime is: " );
22       Console.WriteLine( time.ToUniversalString() );
23       Console.Write( "Standard time after SetTime is: " );
24       Console.WriteLine( time.ToString() );
25       Console.WriteLine(); // output a blank line
26
27       // set time with invalid values; output updated time
28       time.SetTime( 99, 99, 99 );
29       Console.WriteLine( "After attempting invalid settings:" );
30       Console.Write( "Universal time: " );
31       Console.WriteLine( time.ToUniversalString() );
32       Console.Write( "Standard time: " );
33       Console.WriteLine( time.ToString() );
34    } // end Main
35 } // end class Time1Test
```

Time1Test.cs

(2 of 2)

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after SetTime is: 13:27:06
Standard time after SetTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 10.2 | `Time1` object used in an application. (Part 2 of 2.)

10.2 `Time` Class Case Study (Cont.)

Notes on the `Time1` Class Declaration

- The actual data representation used within the class is of no concern to the class's clients, so fields are noramlly declared private.
- Clients could use the same `public` methods and properties to get the same results without being aware a change in the internal representation.

Software Engineering Observation 10.2

Classes simplify programming because the client can use only the `public` members exposed by the class. Such members are usually client oriented rather than implementation oriented. Clients are neither aware of, nor involved in, a class's implementation. Clients generally care about *what* the class does but not how the class does it. (Clients do, of course, care that the class operates correctly and efficiently.)

10.2 `Time` Class Case Study (Cont.)

Software Engineering Observation 10.3

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other application parts become dependent on class-implementation details.

10.3 Controlling Access to Members

- The access modifiers `public` and `private` control access to a class's variables and methods.
- The primary purpose of `public` methods is to present to the class's clients a view of the services the class provides.
- Clients of the class need not be concerned with how the class accomplishes its tasks.
- A class's `private` variables, properties and methods are not directly accessible to the class's clients.

- Figure 10.3 demonstrates that private class members are not directly accessible outside the class.

MemberAccessTest.cs

(1 of 2)

```
1  // Fig. 10.3: MemberAccessTest.cs
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void Main( string[] args )
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7; // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // end Main
13 } // end class MemberAccessTest
```

Attempts to directly access `private` instance variables result in compilation errors.

Error List

3 Errors | 0 Warnings | 0 Messages

	Description	File	Line	Column	Project
1	'Time1.hour' is inaccessible due to its protection level	MemberAccessTest.cs	9	12	MemberAccessTest
2	'Time1.minute' is inaccessible due to its protection level	MemberAccessTest.cs	10	12	MemberAccessTest
3	'Time1.second' is inaccessible due to its protection level	MemberAccessTest.cs	11	12	MemberAccessTest

Fig. 10.3 | Private members of class `Time1` are not accessible.

MemberAccessTest.cs

(2 of 2)

Common Programming Error 10.1

An attempt by a method that is not a member of a class to access a `private` member of that class is a compilation error.

- **Members of a class—for instance, properties, methods and instance variables—have `private` access by default.**

- Every object can access a reference to itself with keyword `this`.
- When a non-`static` method is called, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods.
- As you'll see in Fig. 10.4, you can also use keyword `this` *explicitly* in a non-`static` method's body.

`ThisTest.cs`

(1 of 3)

```
1  // Fig. 10.4: ThisTest.cs
2  // this used implicitly and explicitly to refer to members of an object.
3  using System;
4
5  public class ThisTest
6  {
7     public static void Main( string[] args )
8     {
9        SimpleTime time = new SimpleTime( 15, 30, 19 );
10       Console.WriteLine( time.BuildString() );
11    } // end Main
12 } // end class ThisTest
13
```

Fig. 10.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 3.)

ThisTest.cs

(2 of 3)

```
14 // class SimpleTime demonstrates the "this" reference
15 public class SimpleTime
16 {
17    private int hour; // 0-23
18    private int minute; // 0-59
19    private int second; // 0-59
20
21    // if the constructor uses parameter names identical to
22    // instance-variable names, the "this" reference is
23    // required to distinguish between names
24    public SimpleTime( int hour, int minute, int second )
25    {
26       this.hour = hour; // set "this" object's hour instance variable
27       this.minute = minute; // set "this" object's minute
28       this.second = second; // set "this" object's second
29    } // end SimpleTime constructor
30
31    // use explicit and implicit "this" to call ToUniversalString
32    public string BuildString()
```

If the constructor's parameter names are identical to the class's instance-variable names, so they hide the corresponding instance variables.

You can use the this reference to refer to hidden instance variables explicitly.

Fig. 10.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 2 of 3.)

ThisTest.cs

(3 of 3)

```
33    {
34       return string.Format( "{0,24}: {1}\n{2,24}: {3}",
35          "this.ToUniversalString()", this.ToUniversalString(),
36          "ToUniversalString()", ToUniversalString() );
37    } // end method BuildString
38
39    // convert to string in universal-time format (HH:MM:SS)
40    public string ToUniversalString()
41    {
42       // "this" is not required here to access instance variables,
43       // because method does not have local variables with same
44       // names as instance variables
45       return string.Format( "{0:D2}:{1:D2}:{2:D2}",
46          this.hour, this.minute, this.second );
47    } // end method ToUniversalString
48 } // end class SimpleTime
```

If a member is not hidden, the this keyword is implied, but can be included explicitly.

```
this.ToUniversalString(): 15:30:19
     ToUniversalString(): 15:30:19
```

Fig. 10.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 3 of 3.)

10.4 Referring to the Current Object's Members with the `this` Reference (Cont.)

- If the constructor's parameter names are identical to the class's instance-variable names, so they hide the corresponding instance variables.
- You can use the `this` reference to refer to hidden instance variables explicitly.
- If a member is not hidden, the this keyword is implied, but can be included explicitly.

10.4 Referring to the Current Object's Members with the `this` Reference (Cont.)

Common Programming Error 10.2

It is often a logic error when a method contains a parameter or local variable that has the same name as an instance variable of the class. In such a case, use reference `this` if you wish to access the instance variable of the class—otherwise, the method parameter or local variable will be referenced.

Error-Prevention Tip 10.1

Avoid method-parameter names or local-variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs.

10.4 Referring to the Current Object's Members with the `this` Reference (Cont.)

Performance Tip 10.1

C# conserves memory by maintaining only one copy of each method per class—this method is invoked by every object of the class. Each object, on the other hand, has its own copy of the class's instance variables (i.e., non-`static` variables). Each method of the class implicitly uses the `this` reference to determine the specific object of the class to manipulate.

10.5 Indexers

- A class that encapsulates lists of data can use keyword `this` to define property-like class members called **indexers** that allow array-style indexed access to lists of elements.
- You can define both integer indices and noninteger indices.
- Indexers can return any type, even one that is different from the type of the underlying data.
- Unlike properties, for which you can choose an appropriate property name, indexers must be defined with keyword `this`.

10.5 Indexers (Cont.)

- Indexers have the general form:

accessModifier returnType `this`[*IndexType1 name1* `,` *IndexType2 name2* `,` *...*]

```
{
    get
    {
        // use name1, name2, ... here to get data
    }
    set
    {
        // use name1, name2, ... here to set data
    }
}
```

- The *IndexType* parameters are accessible to the `get` and `set` accessors.

10.5 Indexers (Cont.)

- The accessors define how to use the index (or indices) to retrieve or modify the appropriate data member.
- The indexer's `get` accessor must `return` a value of type *returnType*.
- As in properties, the `set` accessor can use the implicit parameter `value` to reference the value that should be assigned to the element.

Common Programming Error 10.3

Declaring indexers as `static` is a syntax error.

- Class `Box` (Fig. 10.5) represents a box with a length, a width and a height.

Box.cs

(1 of 3)

```
1  // Fig. 10.5: Box.cs
2  // Box class definition represents a box with length,
3  // width and height dimensions with indexers.
4  public class Box
5  {
6     private string[] names = { "length", "width", "height" };
7     private double[] dimensions = new double[ 3 ];
8
9     // constructor
10    public Box( double length, double width, double height )
11    {
12       dimensions[ 0 ] = length;
13       dimensions[ 1 ] = width;
14       dimensions[ 2 ] = height;
15    }
16
17    // indexer to access dimensions by integer index number
```

Fig. 10.5 | `Box` class definition represents a box with length, width and height dimensions with indexers. (Part 1 of 3.)

```
18    public double this[ int index ]
19    {
20       get
21       {
22          // validate index to get
23          if ( ( index < 0 ) || ( index >= dimensions.Length ) )
24             return -1;
25          else
26             return dimensions[ index ];
27       } // end get
28       set
29       {
30          if ( index >= 0 && index < dimensions.Length )
31             dimensions[ index ] = value;
32       } // end set
33    } // end numeric indexer
34
35    // indexer to access dimensions by their string names
36    public double this[ string name ]
37    {
38       get
39       {
```

Box.cs

(2 of 3)

Manipulate the array by index.

Manipulate the array by dimension name.

Fig. 10.5 | Box class definition represents a box with length, width and height dimensions with indexers. (Part 2 of 3.)

```
40          // locate element to get
41          int i = 0;
42          while ( ( i < names.Length ) &&
43             ( name.ToLower() != names[ i ] ) )
44             i++;
45
46          return ( i == names.Length ) ? -1 : dimensions[ i ];
47       } // end get
48       set
49       {
50          // locate element to set
51          int i = 0;
52          while ( ( i < names.Length ) &&
53             ( name.ToLower() != names[ i ] ) )
54             i++;
55
56          if ( i != names.Length )
57             dimensions[ i ] = value;
58       } // end set
59    } // end string indexer
60 } // end class Box
```

Box.cs

(3 of 3)

Manipulate the array by dimension name.

Fig. 10.5 | `Box` class definition represents a box with length, width and height dimensions with indexers. (Part 3 of 3.)

- Indexers can be overloaded like methods.
- Class `BoxTest` (Fig. 10.6) manipulates the `private` data members of class `Box` through `Box`'s indexers.

`BoxTest.cs`

(1 of 3)

```
1  // Fig. 10.6: BoxTest.cs
2  // Indexers provide access to a Box object's members.
3  using System;
4
5  public class BoxTest
6  {
7     public static void Main( string[] args )
8     {
9        // create a box
10       Box box = new Box( 30, 30, 30 );
11
12       // show dimensions with numeric indexers
13       Console.WriteLine( "Created a box with the dimensions:" );
14       Console.WriteLine( "box[ 0 ] = {0}", box[ 0 ] );
15       Console.WriteLine( "box[ 1 ] = {0}", box[ 1 ] );
```

Implicitly call the `get` accessor of the indexer to obtain the value of `box`'s `private` instance variable `dimensions[ 0 ]`.

Fig. 10.6 | Indexers provide access to an object's members. (Part 1 of 3.)

BoxTest.cs

(2 of 3)

```
16         Console.WriteLine( "box[ 2 ] = {0}", box[ 2 ] );
17
18         // set a dimension with the numeric indexer
19         Console.WriteLine( "\nSetting box[ 0 ] to 10...\n" );
20         box[ 0 ] = 10;
21
22         // set a dimension with the string indexer
23         Console.WriteLine( "Setting box[ \"width\" ] to 20...\n" );
24         box[ "width" ] = 20;
25
26         // show dimensions with string indexers
27         Console.WriteLine( "Now the box has the dimensions:" );
28         Console.WriteLine( "box[ \"length\" ] = {0}", box[ "length" ] );
29         Console.WriteLine( "box[ \"width\" ] = {0}", box[ "width" ] );
30         Console.WriteLine( "box[ \"height\" ] = {0}", box[ "height" ] );
31      } // end Main
32   } // end class BoxTest
```

Implicitly call the indexer's `set` accessor.

Fig. 10.6 | Indexers provide access to an object's members. (Part 2 of 3.)

BoxTest.cs

(3 of 3)

```
Created a box with the dimensions:
box[ 0 ] = 30
box[ 1 ] = 30
box[ 2 ] = 30

Setting box[ 0 ] to 10...

Setting box[ "width" ] to 20...

Now the box has the dimensions:
box[ "length" ] = 10
box[ "width" ] = 20
box[ "height" ] = 30
```

Fig. 10.6 | Indexers provide access to an object's members. (Part 3 of 3.)

Outline

- **Overloaded constructors** enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide multiple constructor declarations with different signatures.

`Time2.cs`

(1 of 5)

Class Time2 with Overloaded Constructors

- Class `Time2` (Fig. 10.7) contains five overloaded constructors for conveniently initializing its objects in a variety of ways.

```
1  // Fig. 10.7: Time2.cs
2  // Time2 class declaration with overloaded constructors.
3  public class Time2
4  {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // Time2 no-argument constructor: initializes each instance variable
10    // to zero; ensures that Time2 objects start in a consistent state
11    public Time2() : this( 0, 0, 0 ) { }
```

The parameterless constructor passes values of `0` to the constructor with three `int` parameters. The use of the `this` reference as shown here is called a **constructor initializer**.

Fig. 10.7 | `Time2` class declaration with overloaded constructors. (Part 1 of 5.)

```
12
13    // Time2 constructor: hour supplied, minute and second defaulted to 0
14    public Time2( int h ) : this( h, 0, 0 ) { }
15
16    // Time2 constructor: hour and minute supplied, second defaulted to 0
17    public Time2( int h, int m ) : this( h, m, 0 ) { }
18
19    // Time2 constructor: hour, minute and second supplied
20    public Time2( int h, int m, int s )
21    {
22       SetTime( h, m, s ); // invoke SetTime to validate time
23    } // end Time2 three-argument constructor
24
25    // Time2 constructor: another Time2 object supplied
26    public Time2( Time2 time )
27       : this( time.hour, time.minute, time.second ) { }
28
29    // set a new time value using universal time; ensure that
30    // the data remains consistent by setting invalid values to zero
```

Time2.cs

(2 of 5)

Declare a `Time2` constructor with a single `int` parameter representing the `hour`. Pass the given hour and 0's to the three-parameter constructor.

Declare the `Time2` constructor that receives three `int` parameters representing the `hour`, `minute` and `second`. This constructor is used by all of the others.

Fig. 10.7 | `Time2` class declaration with overloaded constructors. (Part 2 of 5.)

Time2.cs

(3 of 5)

```
31    public void SetTime( int h, int m, int s )
32    {
33       Hour = h; // set the Hour property
34       Minute = m; // set the Minute property
35       Second = s; // set the Second property
36    } // end method SetTime
37
38    // Properties for getting and setting
39    // property that gets and sets the hour
40    public int Hour
41    {
42       get
43       {
44          return hour;
45       } // end get
46       // make writing inaccessible outside the class
47       private set
48       {
49          hour = ( ( value >= 0 && value < 24 ) ? value : 0 );
50       } // end set
51    } // end property Hour
```

Fig. 10.7 | `Time2` class declaration with overloaded constructors. (Part 3 of 5.)

Time2.cs

(4 of 5)

```
52
53    // property that gets and sets the minute
54    public int Minute
55    {
56       get
57       {
58          return minute;
59       } // end get
60       // make writing inaccessible outside the class
61       private set
62       {
63          minute = ( ( value >= 0 && value < 60 ) ? value : 0 );
64       } // end set
65    } // end property Minute
66
67    // property that gets and sets the second
68    public int Second
69    {
70       get
71       {
72          return second;
73       } // end get
```

Fig. 10.7 | Time2 class declaration with overloaded constructors. (Part 4 of 5.)

Time2.cs

(5 of 5)

```
74       // make writing inaccessible outside the class
75       private set
76       {
77          second = ( ( value >= 0 && value < 60 ) ? value : 0 );
78       } // end set
79    } // end property Second
80
81    // convert to string in universal-time format (HH:MM:SS)
82    public string ToUniversalString()
83    {
84       return string.Format(
85          "{0:D2}:{1:D2}:{2:D2}", hour, minute, second );
86    } // end method ToUniversalString
87
88    // convert to string in standard-time format (H:MM:SS AM or PM)
89    public override string ToString()
90    {
91       return string.Format( "{0}:{1:D2}:{2:D2} {3}",
92          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
93          minute, second, ( hour < 12 ? "AM" : "PM" ) );
94    } // end method ToString
95 } // end class Time2
```

Fig. 10.7 | `Time2` class declaration with overloaded constructors. (Part 5 of 5.)

10.6 `Time` Class Case Study: Overloaded Constructors (Cont.)

- Constructor initializers are a popular way to reuse initialization code provided by one of the class's constructors.

Common Programming Error 10.4

A constructor can call methods of the class. Be aware that the instance variables might not yet be in a consistent state, because the constructor is in the process of initializing the object. Using instance variables before they have been initialized properly is a logic error.

Software Engineering Observation 10.4

When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are `private`).

10.6 `Time` Class Case Study: Overloaded Constructors (Cont.)

Notes Regarding Class `Time2`'s Methods, Properties and Constructors

- Consider changing the representation of the time to a single `int` value representing the total number of seconds that have elapsed since midnight.
  - Only the bodies of the methods that access the `private` data directly would need to change.
  - There would be no need to modify the bodies of methods `SetTime`, `ToUniversalString` or `ToString`.

10.6 `Time` Class Case Study: Overloaded Constructors (Cont.)

Software Engineering Observation 10.5

When implementing a method of a class, use the class's properties to access the class's `private` data. This simplifies code maintenance and reduces the likelihood of errors.

- When there is no access modifier before a `get` or `set` accessor, the accessor inherits the access modifier preceding the property name.

Using Class `Time2`'s Overloaded Constructors

- Class `Time2Test` (Fig. 10.8) creates six `Time2` objects to invoke the overloaded `Time2` constructors.

`Time2Test.cs`

(1 of 3)

```
1  // Fig. 10.8: Time2Test.cs
2  // Overloaded constructors used to initialize Time2 objects.
3  using System;
4
5  public class Time2Test
6  {
7     public static void Main( string[] args )
8     {
9        Time2 t1 = new Time2(); // 00:00:00
10       Time2 t2 = new Time2( 2 ); // 02:00:00
11       Time2 t3 = new Time2( 21, 34 ); // 21:34:00
12       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
13       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
14       Time2 t6 = new Time2( t4 ); // 12:25:42
15
16       Console.WriteLine( "Constructed with:\n" );
17       Console.WriteLine( "t1: all arguments defaulted" );
18       Console.WriteLine( "   {0}", t1.ToUniversalString() ); // 00:00:00
19       Console.WriteLine( "   {0}\n", t1.ToString() ); // 12:00:00 AM
```

Fig. 10.8 | Overloaded constructors used to initialize `Time2` objects. (Part 1 of 3.)

```
20
21       Console.WriteLine(
22          "t2: hour specified; minute and second defaulted" );
23       Console.WriteLine( "   {0}", t2.ToUniversalString() ); // 02:00:00
24       Console.WriteLine( "   {0}\n", t2.ToString() ); // 2:00:00 AM
25
26       Console.WriteLine(
27          "t3: hour and minute specified; second defaulted" );
28       Console.WriteLine( "   {0}", t3.ToUniversalString() ); // 21:34:00
29       Console.WriteLine( "   {0}\n", t3.ToString() ); // 9:34:00 PM
30
31       Console.WriteLine( "t4: hour, minute and second specified" );
32       Console.WriteLine( "   {0}", t4.ToUniversalString() ); // 12:25:42
33       Console.WriteLine( "   {0}\n", t4.ToString() ); // 12:25:42 PM
34
35       Console.WriteLine( "t5: all invalid values specified" );
36       Console.WriteLine( "   {0}", t5.ToUniversalString() ); // 00:00:00
37       Console.WriteLine( "   {0}\n", t5.ToString() ); // 12:00:00 AM
38
39       Console.WriteLine( "t6: Time2 object t4 specified" );
40       Console.WriteLine( "   {0}", t6.ToUniversalString() ); // 12:25:42
41       Console.WriteLine( "   {0}", t6.ToString() ); // 12:25:42 PM
42    } // end Main
43 } // end class Time2Test
43 } // end class Time2Test
```

Time2Test.cs

(2 of 3)

Fig. 10.8 | Overloaded constructors used to initialize Time2 objects. (Part 1 of 3.)

Time2Test.cs

(3 of 3)

```
Constructed with:

t1: all arguments defaulted
   00:00:00
   12:00:00 AM

t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM

t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM

t4: hour, minute and second specified
   12:25:42
   12:25:42 PM

t5: all invalid values specified
   00:00:00
   12:00:00 AM

t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

Fig. 10.8 | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 3.)

10.7 Default and Parameterless Constructors

- Every class must have at least one constructor.If you do not provide any constructors in a class's declaration, the compiler creates a default constructor that takes no arguments when it is invoked.
- The compiler will not create a default constructor for a class that explicitly declares at least one constructor.
- If you have declared a constructor, but want to be able to invoke the constructor with no arguments, you must declare a parameterless constructor.

10.7 Default and Parameterless Constructors (Cont.)

Common Programming Error 10.5

If a class has constructors, but none of the `public` constructors are parameterless constructors, and an application attempts to call a parameterless constructor to initialize an object of the class, a compilation error occurs. A constructor can be called with no arguments only if the class does not have any constructors (in which case the default constructor is called) or if the class has a `public` parameterless constructor.

Common Programming Error 10.6

Only constructors can have the same name as the class. Declaring a method, property or field with the same name as the class is a compilation error.

S.O.L.I.D The First 5 Principles of Object Oriented Design

S.O.L.I.D is an acronym for the **first five object-oriented design**

- ✓ **Single-responsibility Principle**
- ✓ **Open-closed Principle**
- ✓ **Liskov substitution principle**
- ✓ **Interface segregation principle**
- ✓ **Dependency Inversion principle**

Single-responsibility Principle

A class should have one and only one reason to change, meaning that a class should have only one job.

Example

Consider a class that **compiles** and **prints** a report. It **may change for two reasons**.

First, the content of the report can change.

Second, the format of the report can change.

The **Single responsibility principle says that these two aspects of the problem are really two separate responsibilities and should therefore be in separate classes**. It would be a **bad design** to **couple two things that change for different reasons** at different times.

S.O.L.I.D

Open-closed Principle

Objects or entities should be open for extension, but closed for modification. This simply means that a class should be easily extendable without modifying the class itself.

Liskov substitution principle

Let `q(x)` be a property provable about objects of `x` of type `T`. Then `q(y)` should be provable for objects y of type `S` where `S` is a subtype of `T`. All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

S.O.L.I.D

Interface segregation principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use. This simply means that a class should not implement an interface, if its semantics does not support its functionality.

Dependency Inversion principle

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions . This principle allows for decoupling the OOD.

Software Engineering Observation

The reason it is important to keep a class focused on a **single concern** is that it **makes the class more robust**. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break, if it is part of the same class.

Composition

A class can have references to objects of other classes as members.
This is called **composition** and is sometimes referred to as a ***HAS-A relationship***.

Reference types:

- ✓ **Mutable**
- ✓ **Immutable**

Composition

In object-oriented and functional programming, an immutable object (unchangeable object) is an object whose state cannot be modified after it is created. This is in contrast to a mutable object (changeable object), which can be modified after it is created. In some cases, an object is considered immutable even if some internally used attributes change but the object's state appears to be unchanging from an external point of view.

Composition

Strings and other concrete objects are typically expressed as immutable objects to improve readability and run time efficiency in object-oriented programming. Immutable objects are also useful because they are inherently thread-safe. Other benefits are that they are simpler to understand and reason about and offer higher security than mutable objects

Composition

To define a simple immutable class follow the below mentioned rules

1. Don't provide "`set`" properties — methods that modify fields or objects referred to by fields.
2. Make all fields `readonly` and `private`.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as sealed. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
5. Don't provide methods that modify the mutable objects.
6. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Composition

Composition

- A class can have references to objects of other classes as members
- Sometimes referred to as a *has-a* relationship

Software Engineering Observation

One form of software reuse is composition, in which a class has as members references to objects of other classes.

One form of software reuse is composition, in which a class has as members references to objects of other classes.

`Date.cs`

(1 of 4)

Class `Date` (Fig. 10.9) declares instance variables `month` and `day`, and auto-implemented property `Year` (line 11) to represent a date.

```
1  // Fig. 10.9: Date.cs
2  // Date class declaration.
3  using System;
4
5  public class Date
6  {
7     private int month; // 1-12
8     private int day; // 1-31 based on month
9
```

Fig. 10.9 | `Date` class declaration. (Part 1 of 4.)

Date.cs

(2 of 4)

```
10    // auto-implemented property Year
11    public int Year { get; set; }
12
13    // constructor: use property Month to confirm proper value for month;
14    // use property Day to confirm proper value for day
15    public Date( int theMonth, int theDay, int theYear )
16    {
17       Month = theMonth; // validate month
18       Year = theYear; // could validate year
19       Day = theDay; // validate day
20       Console.WriteLine( "Date object constructor for date {0}", this );
21    } // end Date constructor
22
23    // property that gets and sets the month
24    public int Month
25    {
26       get
27       {
28          return month;
29       } // end get
30       private set // make writing inaccessible outside the class
```

Fig. 10.9 | Date class declaration. (Part 2 of 4.)

Date.cs

(3 of 4)

```
31       {
32          if ( value > 0 && value <= 12 ) // validate month
33             month = value;
34          else // month is invalid
35          {
36             Console.WriteLine( "Invalid month ({0}) set to 1.", value );
37             month = 1; // maintain object in consistent state
38          } // end else
39       } // end set
40    } // end property Month
41
42    // property that gets and sets the day
43    public int Day
44    {
45       get
46       {
47          return day;
48       } // end get
49       private set // make writing inaccessible outside the class
50       {
51          int[] daysPerMonth = { 0, 31, 28, 31, 30, 31, 30,
52                                 31, 31, 30, 31, 30, 31 };
```

Fig. 10.9 | Date class declaration. (Part 3 of 4.)

Date.cs

(4 of 4)

```
53
54          // check if day in range for month
55          if ( value > 0 && value <= daysPerMonth[ Month ] )
56             day = value;
57          // check for leap year
58          else if ( month == 2 && value == 29 &&
59             ( year % 400 == 0 || ( year % 4 == 0 && year % 100 != 0 ) ) )
60             day = value;
61          else
62          {
63             Console.WriteLine( "Invalid day ({0}) set to 1.", value );
64             day = 1; // maintain object in consistent state
65          } // end else
66       } // end set
67    } // end property Day
68
69    // return a string of the form month/day/year
70    public override string ToString()
71    {
72       return string.Format( "{0}/{1}/{2}", month, day, year );
73    } // end method ToString
74 } // end class Date
```

Fig. 10.9 | Date class declaration. (Part 4 of 4.)

- Class Employee (Fig. 10.10) has instance variables `firstName`, `lastName`, `birthDate` and `hireDate`.

`Employee.cs`

(1 of 2)

```
1  // Fig. 10.10: Employee.cs
2  // Employee class with references to other objects.
3  public class Employee
4  {
5     private string firstName;
6     private string lastName;
7     private Date birthDate;
8     private Date hireDate;
9
10    // constructor to initialize name, birth date and hire date
11    public Employee( string first, string last,
12       Date dateOfBirth, Date dateOfHire )
13    {
14       firstName = first;
15       lastName = last;
16       birthDate = dateOfBirth;
17       hireDate = dateOfHire;
18    } // end Employee constructor
```

Members `birthDate` and `hireDate` are references to `Date` objects, demonstrating that a class can have as instance variables references to objects of other classes.

Wrong. Encapsulation violated.! Write a **copy constructor** in `Date` and **assign a `Date` object copy** to `birthDate` and `hireDate`. Simialrly, **write properties** for `birthDate` and `hireDate` that **work with copy objects** of `Date`

Fig. 10.10 | `Employee` class with references to other objects. (Part 1 of 2.)

Employee.cs

(2 of 2)

```
19
20    // convert Employee to string format
21    public override string ToString()
22    {
23       return string.Format( "{0}, {1}  Hired: {2}  Birthday: {3}",
24          lastName, firstName, hireDate, birthDate );
25    } // end method ToString
26 } // end class Employee
```

Fig. 10.10 | `Employee` class with references to other objects. (Part 2 of 2.)

- Class `EmployeeTest` (Fig. 10.11) creates two `Date` objects to represent an `Employee`'s birthday and hire date, respectively.

`EmployeeTest.cs`

```
1  // Fig. 10.11: EmployeeTest.cs
2  // Composition demonstration.
3  using System;
4
5  public class EmployeeTest
6  {
7     public static void Main( string[] args )
8     {
9        Date birth = new Date( 7, 24, 1949 );
10       Date hire = new Date( 3, 12, 1988 );
11       Employee employee = new Employee( "Bob", "Blue", birth, hire );
12
13       Console.WriteLine( employee );
14    } // end Main
15 } // end class EmployeeTest
```

Pass the names and two `Date` objects to the `Employee` constructor.

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 10.11 | Composition demonstration.

10.9 Garbage Collection and Destructors

- Every object you create uses various system resources, such as memory.
- In many programming languages, these system resources are reserved for the object's use until they are explicitly released by the programmer.
- If all the references to the object that manages the resource are lost before the resource is explicitly released, it can no longer be released. This is known as a **resource leak**.
- The Common Language Runtime (CLR) uses a **garbage collector** to reclaim the memory occupied by objects that are no longer in use.
- When there are no more references to an object, the object becomes **eligible for destruction**.

10.9 Garbage Collection and Destructors (Cont.)

- Every object has a **destructor** that is invoked by the garbage collector to perform **termination housekeeping** before its memory is reclaimed.
- A destructor's name is the class name, preceded by a tilde, and it has no access modifier in its header.
- After an object's destructor is called, the object becomes **eligible for garbage collection**—the memory for the object can be reclaimed by the garbage collector.
- **Memory leaks** are less likely in C# than languages like C and C++ (but some can still happen in subtle ways).

10.9 Garbage Collection and Destructors (Cont.)

- Other types of resource leaks can occur, for example if an application fails to close a file that it has opened.
- A problem with the garbage collector is that it is not guaranteed to perform its tasks at a specified time. For this reason, destructors are rarely used.

Software Engineering Observation 10.7

A class that uses system resources, such as files on disk, should provide a method to eventually release the resources. Many Framework Class Library classes provide `Close` or `Dispose` methods for this purpose.

10.10 `static` Class Members

- A `static` **variable** is used when only one copy of a particular variable should be shared by all objects of a class.
- A `static` variable represents **classwide information**—all objects of the class share the same piece of data.
- The declaration of a `static` variable begins with the keyword `static`.

Software Engineering Observation 10.8

Use a `static` variable when all objects of a class must use the same copy of the variable.

10.10 `static` Class Members (Cont.)

- The scope of a `static` variable is the body of its class.
- A class's `public static` members can be accessed by qualifying the member name with the class name and the member access (`.`) operator, as in `Math.PI`.
- A class's `private static` class members can be accessed only through the methods and properties of the class.
- `static` class members exist even when no objects of the class exist—they are available as soon as the class is loaded into memory at execution time.
- To access a `private static` member from outside its class, a `public static` method or property can be provided.

10.10 static Class Members (Cont.)

Common Programming Error 10.7

It is a compilation error to access or invoke a `static` member by referencing it through an instance of the class, like a non-`static` member.

Software Engineering Observation 10.9

Static variables and methods exist, and can be used, even if no objects of that class have been instantiated.

- Class Employee (Fig. 10.12) declares `private static` variable `count` and `public static` property `Count`.

`Employee.cs`

(1 of 2)

```
1  // Fig. 10.12: Employee.cs
2  // Static variable used to maintain a count of the number of
3  // Employee objects that have been created.
4  using System;
5
6  public class Employee
7  {
8     private static int count = 0; // number of objects in memory
9
10    // read-only auto-implemented property FirstName
11    public string FirstName { get; private set; }
12
13    // read-only auto-implemented property LastName
14    public string LastName { get; private set; }
15
```

If a `static` variable is not initialized, the compiler assigns a default value to the variable.

Fig. 10.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 1 of 2.)

```
16    // initialize employee, add 1 to static count and
17    // output string indicating that constructor was called
18    public Employee( string first, string last )
19    {
20       FirstName = first;
21       LastName = last;
22       count++; // increment static count of employees
23       Console.WriteLine( "Employee constructor: {0} {1}; count = {2}",
24          FirstName, LastName, Count );
25    } // end Employee constructor
26
27    // read-only property that gets the employee count
28    public static int Count
29    {
30       get
31       {
32          return count;
33       } // end get
34    } // end property Count
35 } // end class Employee
```

Employee.cs

(2 of 2)

Variable `count` maintains a count of the number of objects of class `Employee` that have been created.

When no objects of class `Employee` exist, member `count` can only be referenced through a call to `public static` property `Count`.

Fig. 10.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 2 of 2.)

- If a `static` variable is not initialized, the compiler assigns a default value to the variable.

- `EmployeeTest` method `Main` (Fig. 10.13) instantiates two `Employee` objects.

`EmployeeTest.cs`

(1 of 2)

```
1  // Fig. 10.13: EmployeeTest.cs
2  // Static member demonstration.
3  using System;
4
5  public class EmployeeTest
6  {
7     public static void Main( string[] args )
8     {
9        // show that count is 0 before creating Employees
10       Console.WriteLine( "Employees before instantiation: {0}",
11          Employee.Count );
12
13       // create two Employees; count should become 2
14       Employee e1 = new Employee( "Susan", "Baker" );
15       Employee e2 = new Employee( "Bob", "Blue" );
16
17       // show that count is 2 after creating two Employees
18       Console.WriteLine( "\nEmployees after instantiation: {0}",
19          Employee.Count );
```

Fig. 10.13 | `static` member demonstration. (Part 1 of 2.)

EmployeeTest.cs

(2 of 2)

```
20
21       // get names of Employees
22       Console.WriteLine( "\nEmployee 1: {0} {1}\nEmployee 2: {2} {3}\n",
23          e1.FirstName, e1.LastName,
24          e2.FirstName, e2.LastName );
25
26       // in this example, there is only one reference to each Employee,
27       // so the following statements cause the CLR to mark each
28       // Employee object as being eligible for garbage collection
29       e1 = null; // good practice: mark object e1 no longer needed
30       e2 = null; // good practice: mark object e2 no longer needed
31    } // end Main
32 } // end class EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation: 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Fig. 10.13 | `static` member demonstration. (Part 2 of 2.)

10.10 static Class Members (Cont.)

- `string` objects in C# are immutable—they cannot be modified after they are created. Therefore, it is safe to have many references to one `string` object.
- String-concatenation operations result in a new `string` object containing the concatenated values. The original `string` objects are not modified.
- C# does not guarantee when, or even whether, the garbage collector will execute.
- When the garbage collector does run, it is possible that no objects or only a subset of the eligible objects will be collected.

10.10 static Class Members (Cont.)

- Force garbage collection

```
System.GC.Collect();
// wait until collection completes
System.GC.WaitForPendingFinalizers();
Console.WriteLine("\nEmployees after instantiation: {0}
 ", Employee.Count);
```

- Use the Finalizer to decrement the actual object Count

```
// in class Employee add
~Employee()
   {
     count--;
     Console.WriteLine( "Employee constructor: {0} {1};
            count = {2}", FirstName, LastName, Count );
   }
```

10.10 static Class Members (Cont.)

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation: 2

Employee 1: Susan Baker
Employee 2: Bob Blue

Employee constructor: Bob Blue; count = 1
Employee constructor: Susan Baker; count = 0

Employees after instantiation: 0
Press any key to continue . . .
```

10.10 static Class Members (Cont.)

- A method declared `static` cannot access non-`static` class members directly, because a `static` method can be called even when no objects of the class exist.
- The `this` reference cannot be used in a `static` method.

Common Programming Error 10.8

A compilation error occurs if a `static` method calls an instance (non-`static`) method in the same class by using only the method name. Similarly, a compilation error occurs if a `static` method attempts to access an instance variable in the same class by using only the variable name.

Common Programming Error 10.9

Referring to the `this` reference in a `static` method is a syntax error.

10.11 `readonly` Instance Variables

- The **principle of least privilege** states that code should be granted only the amount of privilege and access needed to accomplish its designated task, but no more.
- Constants declared with `const` must be initialized to a constant value when they are declared.
- C# provides keyword `readonly` to specify that an instance variable of an object is not modifiable and that any attempt to modify it after the object is constructed is an error.
- Like constants, `readonly` variables are declared with all capital letters by convention
- `readonly` instance variables can be initialized when they are declared, but this is not required.

10.11 `readonly` Instance Variables (Cont.)

- A `readonly` instance variable doesn't become unmodifiable until after the constructor completes execution.

Software Engineering Observation 10.10

Declaring an instance variable as `readonly` helps enforce the principle of least privilege. If an instance variable should not be modified after the object is constructed, declare it to be `readonly` to prevent modification.

- Members that are declared as `const` must be assigned values at compile time, whereas members declared with keyword `readonly`, can be initialized at execution time.
- Variables that are `readonly` can be initialized with expressions that are not contsants, such as an array initializer or a method call.

- Class `Increment` (Fig. 10.14) contains a `readonly` instance variable of type `int` named `INCREMENT`.

`Increment.cs`

(1 of 2)

```
1  // Fig. 10.14: Increment.cs
2  // readonly instance variable in a class.
3  public class Increment
4  {
5     // readonly instance variable (uninitialized)
6     private readonly int INCREMENT;
7     private int total = 0; // total of all increments
8
9     // constructor initializes readonly instance variable INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // initialize readonly variable (once)
13    } // end Increment constructor
14
```

The `readonly` variable is not initialized in its declaration

Fig. 10.14 | `readonly` instance variable in a class. (Part 1 of 2.)

Increment.cs

(2 of 2)

```
15    // add INCREMENT to total
16    public void AddIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // end method AddIncrementToTotal
20
21    // return string representation of an Increment object's data
22    public override string ToString()
23    {
24       return string.Format( "total = {0}", total );
25    } // end method ToString
26 } // end class Increment
```

Fig. 10.14 | `readonly` instance variable in a class. (Part 2 of 2.)

- If a class provides multiple constructors, every constructor should initialize a `readonly` variable.
- If a constructor does not initialize the `readonly` variable, the variable receives the same default value as any other instance variable, and the compiler generates a warning.
- Application class `IncrementTest` (Fig. 10.15) demonstrates class `Increment`.

`IncrementTest.cs`

(1 of 3)

```
1  // Fig. 10.15: IncrementTest.cs
2  // readonly instance variable initialized with a constructor argument.
3  using System;
4
5  public class IncrementTest
6  {
7     public static void Main( string[] args )
8     {
9        Increment incrementer = new Increment( 5 );
10
11       Console.WriteLine( "Before incrementing: {0}\n", incrementer );
12
```

Fig. 10.15 | `readonly` instance variable initialized with a constructor argument. (Part 1 of 2.)

IncrementTest.cs

(2 of 3)

```
13       for ( int i = 1; i <= 3; i++ )
14       {
15          incrementer.AddIncrementToTotal();
16          Console.WriteLine( "After increment {0}: {1}", i, incrementer );
17       } // end for
18    } // end Main
19 } // end class IncrementTest
```

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Fig. 10.15 | `readonly` instance variable initialized with a constructor argument. (Part 2 of 2.)

Common Programming Error 10.10

Attempting to modify a `readonly` instance variable anywhere but in its declaration or the object's constructors is a compilation error.

Error-Prevention Tip 10.2

Attempts to modify a `readonly` instance variable are caught at compilation time rather than causing execution-time errors. It is always preferable to get bugs out at compile time, if possible, rather than allowing them to slip through to execution time (where studies have found that repairing is often many times more costly).

Software Engineering Observation 10.11

If a `readonly` instance variable is initialized to a constant only in its declaration, it is not necessary to have a separate copy of the instance variable for every object of the class. The variable should be declared `const` instead. Constants declared with `const` are implicitly `static`, so there will only be one copy for the entire class.

10.12 Software Reusability

- Programmers concentrate on crafting new classes and reusing existing classes.
- Software reusability speeds the development of powerful, high-quality software.
- **Rapid application development (RAD)** is of great interest today.
- Microsoft provides C# programmers with thousands of classes in the .NET Framework Class Library to help them implement C# applications.
- To take advantage of C#'s many capabilities, it is essential that programmers familiarize themselves with the variety of classes in the .NET Framework.

10.12 Software Reusability (Cont.)

Good Programming Practice 10.1

Avoid reinventing the wheel. Study the capabilities of the Framework Class Library. If the library contains a class that meets your application's requirements, use that class rather than create your own.

10.13 Data Abstraction and Encapsulation

- Classes normally hide the details of their implementation from their clients. This is called **information hiding**.
- The client cares about what functionality a class offers, not about how that functionality is implemented. This concept is referred to as **data abstraction**.
- Although programmers might know the details of a class's implementation, they should not write code that depends on these details as the details may later change.
- C# and the object-oriented style of programming elevate the importance of data.
- The primary activities of object-oriented programming in C# are the creation of types (e.g., classes) and the expression of the interactions among objects of those types.

10.13 Data Abstraction and Encapsulation (Cont.)

- **Abstract data types (ADTs)** improve the application-development process.
- Types like `int`, `double`, and `char` are all examples of abstract data types.
- ADTs are representations of real-world concepts to some satisfactory level of precision within a computer system.
- An ADT actually captures two notions: a **data representation** and the **operations** that can be performed on that data.
- C# programmers use classes to implement abstract data types.

Software Engineering Observation 10.12

Programmers create types through the class mechanism. New types can be designed to be as convenient to use as the simple types. Although the language is easy to extend via new types, the programmer cannot alter the base language itself.

10.13 Data Abstraction and Encapsulation (Cont.)

- Clients place items in a **queue** one at a time via an *enqueue* operation, then get them back one at a time via a *dequeue* operation.
- A queue returns items in **first-in, first-out (FIFO)** order, which means that the first item inserted in a queue is the first item removed from the queue.
- Conceptually, a queue can become infinitely long, but real queues are finite.
- Only the queue ADT has access to its internal data.

10.14 `Time` Class Case Study: Creating Class Libraries

- As applications become more complex, namespaces help you manage the complexity of application components.
- Class libraries and namespaces also facilitate software reuse by enabling applications to add classes from other namespaces.

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

Steps for Declaring and Using a Reusable Class

- Before a class can be used in multiple applications, it must be placed in a class library to make it reusable.
- The steps for creating a reusable class are:
  - Declare a `public` class. If the class is not `public`, it can be used only by other classes in the same assembly.
  - Choose a namespace name and add a **`namespace` declaration** to the source-code file for the reusable class declaration.
  - Compile the class into a class library.
  - Add a reference to the class library in an application.
  - Specify a `using` directive for the namespace of the reusable class and use the class.

Step 1: Creating a `public` Class

- We use the `public` class `Time1` declared in Fig. 10.1. No modifications have been made to the implementation of the class.

`Time1.cs`

(1 of 2)

Step 2: Adding the `namespace` Declaration

- The new version of the `Time1` class with the `namespace` declaration is shown in Fig. 10.16.

```
1  // Fig. 10.16: Time1.cs
2  // Time1 class declaration in a namespace.
3  namespace Chapter10
4  {
5     public class Time1
6     {
7        private int hour; // 0 - 23
8        private int minute; // 0 - 59
9        private int second; // 0 - 59
10
11       // set a new time value using universal time; ensure that
12       // the data remains consistent by setting invalid values to zero
13       public void SetTime( int h, int m, int s )
14       {
```

Declares a `namespace` named `Chapter10`.

Fig. 10.16 | `Time1` class declaration in a `namespace`. (Part 1 of 2.)

Time1.cs

(2 of 2)

```
15          hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
16          minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17          second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18       } // end method SetTime
19
20       // convert to string in universal-time format (HH:MM:SS)
21       public string ToUniversalString()
22       {
23          return string.Format( "{0:D2}:{1:D2}:{2:D2}",
24             hour, minute, second );
25       } // end method ToUniversalString
26
27       // convert to string in standard-time format (H:MM:SS AM or PM)
28       public override string ToString()
29       {
30          return string.Format( "{0}:{1:D2}:{2:D2} {3}",
31             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
32             minute, second, ( hour < 12 ? "AM" : "PM" ) );
33       } // end method ToString
34    } // end class Time1
35 } // end namespace Chapter10
```

Fig. 10.16 | `Time1` class declaration in a `namespace`. (Part 2 of 2.)

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

- Placing a class inside a `namespace` declaration indicates that the class is part of the specified namespace.
- The `namespace` name is part of the fully qualified class name, so the name of class `Time1` is actually `Chapter10.Time1`.
- You can use this fully qualified name in your applications, or you can write a `using` directive and use its **simple name** (`Time1`) in the application.
- If another namespace also contains a `Time1` class, use fully qualified class names to prevent a **name conflict** (also called a **name collision**).

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

- Most language elements must appear inside the braces of a type declaration (e.g., classes and enumerations).
- Some exceptions are `namespace` declarations, `using` directives, comments and C# attributes.
- Only class declarations declared `public` will be reusable by clients of the class library.
- Non-`public` classes are typically placed in a library to support the `public` reusable classes in that library.

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

Step 3: Compiling the Class Library

- To create a class library in Visual C# Express, we must create a new project and choose **Class Library** from the list of templates, as shown in Fig. 10.17.

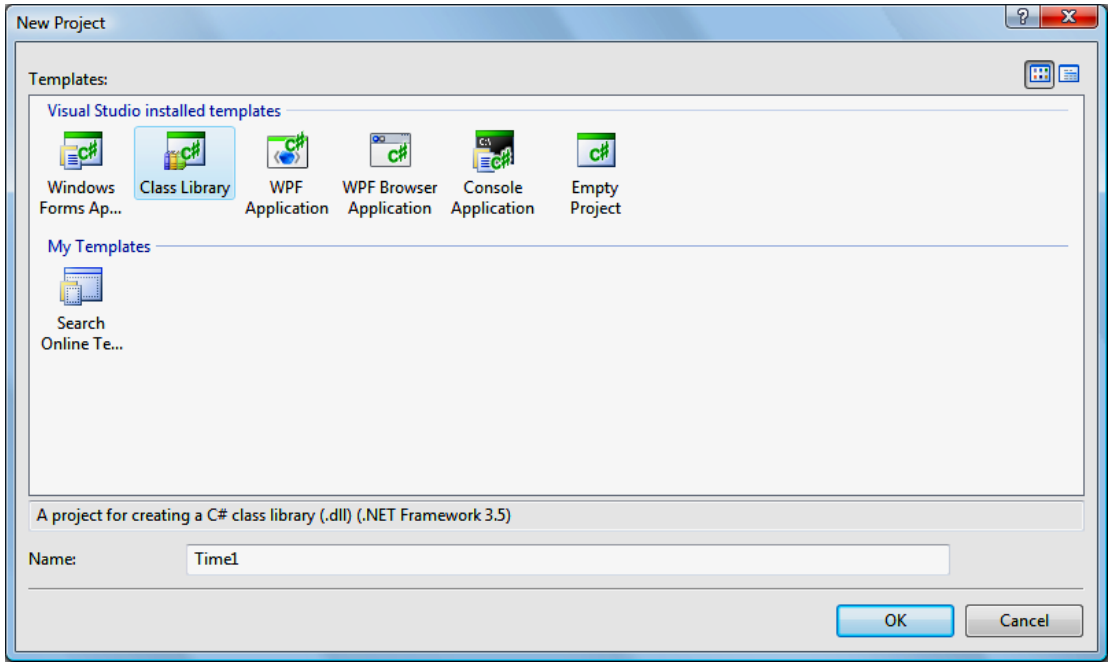

Fig. 10.17 | Creating a **Class Library** Project.

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

- Then add the code for the class, including the namespace declaration, into the project.
- When you compile a Class Library project, the compiler creates a **`.dll` file**, known as a **dynamically linked library**—a type of assembly that you can reference from other applications.

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

Step 4: Adding a Reference to the Class Library

- The library can now be referenced from any application by indicating to the Visual C# Express IDE where to find the class library file.
- To add a reference to your class library to a project as shown in Fig. 10.18, right-click the project name in the **Solution Explorer** window and select **Add Reference...**.

10.14 `Time` Class Case Study: Creating Class Libraries (Cont.)

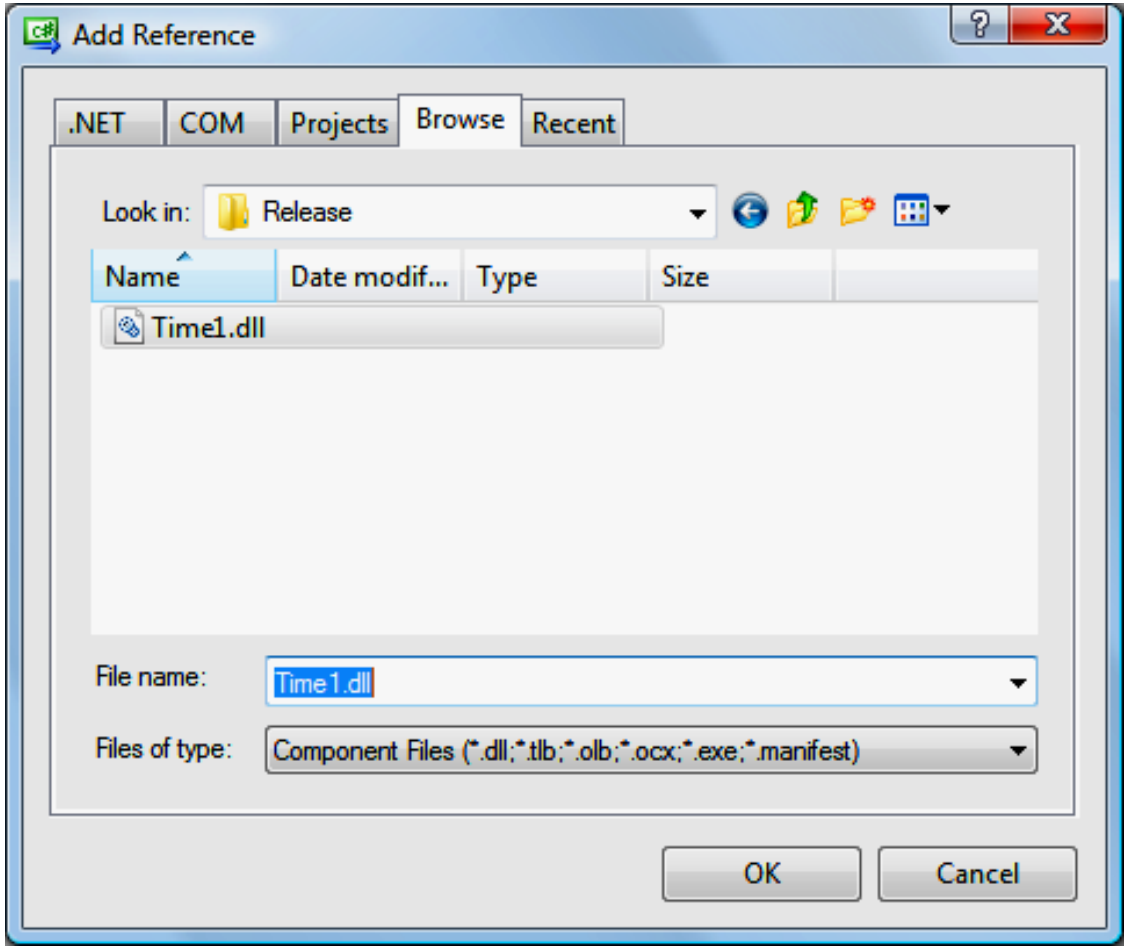

Fig. 10.18 | Adding a Reference.

Step 5: Using the Class from an Application

- Add a new code file to your application and enter the code for class `Time1NamespaceTest` (Fig. 10.19).

`Time1NamespaceTest.cs`

(1 of 3)

```
1  // Fig. 10.19: Time1NamespaceTest.cs
2  // Time1 object used in an application.
3  using Chapter10;
4  using System;
5
6  public class Time1NamespaceTest
7  {
8     public static void Main( string[] args )
9     {
10       // create and initialize a Time1 object
11       Time1 time = new Time1(); // calls Time1 constructor
12
13       // output string representations of the time
14       Console.Write( "The initial universal time is: " );
15       Console.WriteLine( time.ToUniversalString() );
16       Console.Write( "The initial standard time is: " );
17       Console.WriteLine( time.ToString() );
18       Console.WriteLine(); // output a blank line
19
```

Specify that we'd like to use the class(es) of namespace `Chapter10` in this file.

Fig. 10.19 | `Time1` object used in an application. (Part 1 of 2.)

```
20       // change time and output updated time
21       time.SetTime( 13, 27, 6 );
22       Console.Write( "Universal time after SetTime is: " );
23       Console.WriteLine( time.ToUniversalString() );
24       Console.Write( "Standard time after SetTime is: " );
25       Console.WriteLine( time.ToString() );
26       Console.WriteLine(); // output a blank line
27
28       // set time with invalid values; output updated time
29       time.SetTime( 99, 99, 99 );
30       Console.WriteLine( "After attempting invalid settings:" );
31       Console.Write( "Universal time: " );
32       Console.WriteLine( time.ToUniversalString() );
33       Console.Write( "Standard time: " );
34       Console.WriteLine( time.ToString() );
35    } // end Main
36 } // end class Time1NamespaceTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after SetTime is: 13:27:06
Standard time after SetTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Time1NamespaceTest.cs

(2 of 3)

Fig. 10.19 | `Time1` object used in an application. (Part 2 of 2.)

Time1NamespaceTest.cs

(3 of 3)

- Your `Time1` class can now be used by `Time1NamespaceTest` without adding the `Time1.cs` source-code file to the project.
- A class is in the global namespace of an application if the class's file does not contain a `namespace` declaration.
- A `using` directive allows you to use classes in different namespaces as if they were in the same namespace.

10.15 `internal` Access

- Classes like the ones we've defined so far—called top-level classes—can be declared with only two access modifiers—`public` and `internal`.
- C# also supports nested classes—classes defined inside other classes.
- Nested classes may also be declared `private` or `protected`.
- If there is no access modifier in a class declaration, the class defaults to **`internal` access**.
- Internal access allows the class to be used by all code in the same assembly as the class, but not by code in other assemblies.
- Methods, instance variables and other members of a class declared `internal` are only accessible to all code compiled in the same assembly.

- The application in Fig. 10.20 demonstrates `internal` access.

`InternalAccessTest.cs`

(1 of 3)

```
1  // Fig. 10.20: InternalAccessTest.cs
2  // Members declared internal in a class are accessible by other classes
3  // in the same assembly.
4  using System;
5
6  public class InternalAccessTest
7  {
8     public static void Main( string[] args )
9     {
10       InternalData internalData = new InternalData();
11
12       // output string representation of internalData
13       Console.WriteLine( "After instantiation:\n{0}", internalData );
14
15       // change internal-access data in internalData
16       internalData.number = 77;
17       internalData.message = "Goodbye";
18
```

Fig. 10.20 | Members declared `internal` in a class are accessible by other classes in the same assembly. (Part 1 of 3.)

InternalAccessTest.cs

(2 of 3)

```
19       // output string representation of internalData
20       Console.WriteLine( "\nAfter changing values:\n{0}", internalData );
21    } // end Main
22 } // end class InternalAccessTest
23
24 // class with internal-access instance variables
25 class InternalData
26 {
27    internal int number; // internal-access instance variable
28    internal string message; // internal-access instance variable
29
30    // constructor
31    public InternalData()
32    {
33       number = 0;
34       message = "Hello";
35    } // end InternalData constructor
36
37    // return InternalData object string representation
38    public override string ToString()
```

Fig. 10.20 | Members declared `internal` in a class are accessible by other classes in the same assembly. (Part 2 of 3.)

InternalAccessTest.cs

(3 of 3)

```
39    {
40       return string.Format(
41          "number: {0}; message: {1}", number, message );
42    } // end method ToString
43 } // end class InternalData
```

```
After instantiation:
number: 0; message: Hello

After changing values:
number: 77; message: Goodbye
```

Fig. 10.20 | Members declared `internal` in a class are accessible by other classes in the same assembly. (Part 3 of 3.)

10.16 Class View and Object Browser

*Using the **Class View** Window*

- The **Class View** displays the fields and methods for all classes in a project. To access this feature, select **Class View** from the **View** menu.
- Figure 10.21 shows the **Class View** for the `Time1` project of Fig. 10.1 (class `Time1`) and Fig. 10.2 (class `TimeTest1`).

10.16 Class View and Object Browser (Cont.)

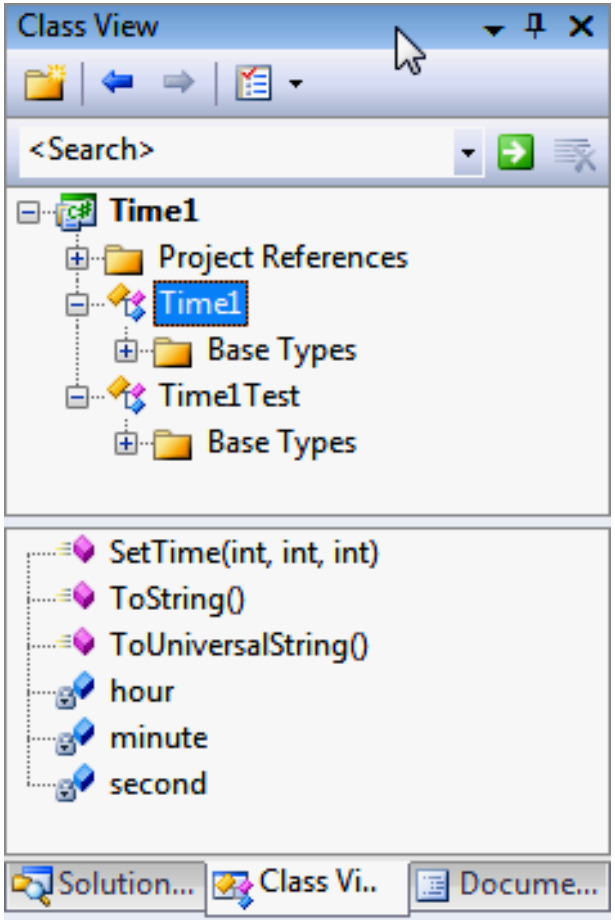

Fig. 10.21 | Class View of class `Time1` (Fig. 10.1) and class `TimeTest` (Fig. 10.2).

10.16 Class View and Object Browser (Cont.)

- The view follows a hierarchical structure, with the project name as the root.
- When a class is selected, its members appear in the lower half of the window.
- Lock icons next to instance variables specify that the variables are `private`.

10.16 Class View and Object Browser (Cont.)

Using the ***Object Browser***

- You can use the **Object Browser** to learn about the functionality provided by a specific class.
- To open the **Object Browser**, select **Other Windows** from the **View** menu and click **Object Browser**.
- Figure 10.22 depicts the **Object Browser** when the user navigates to the `Math` class in namespace `System` in the assembly `mscorlib.dll` (Microsoft Core Library).

10.16 Class View and Object Browser (Cont.)

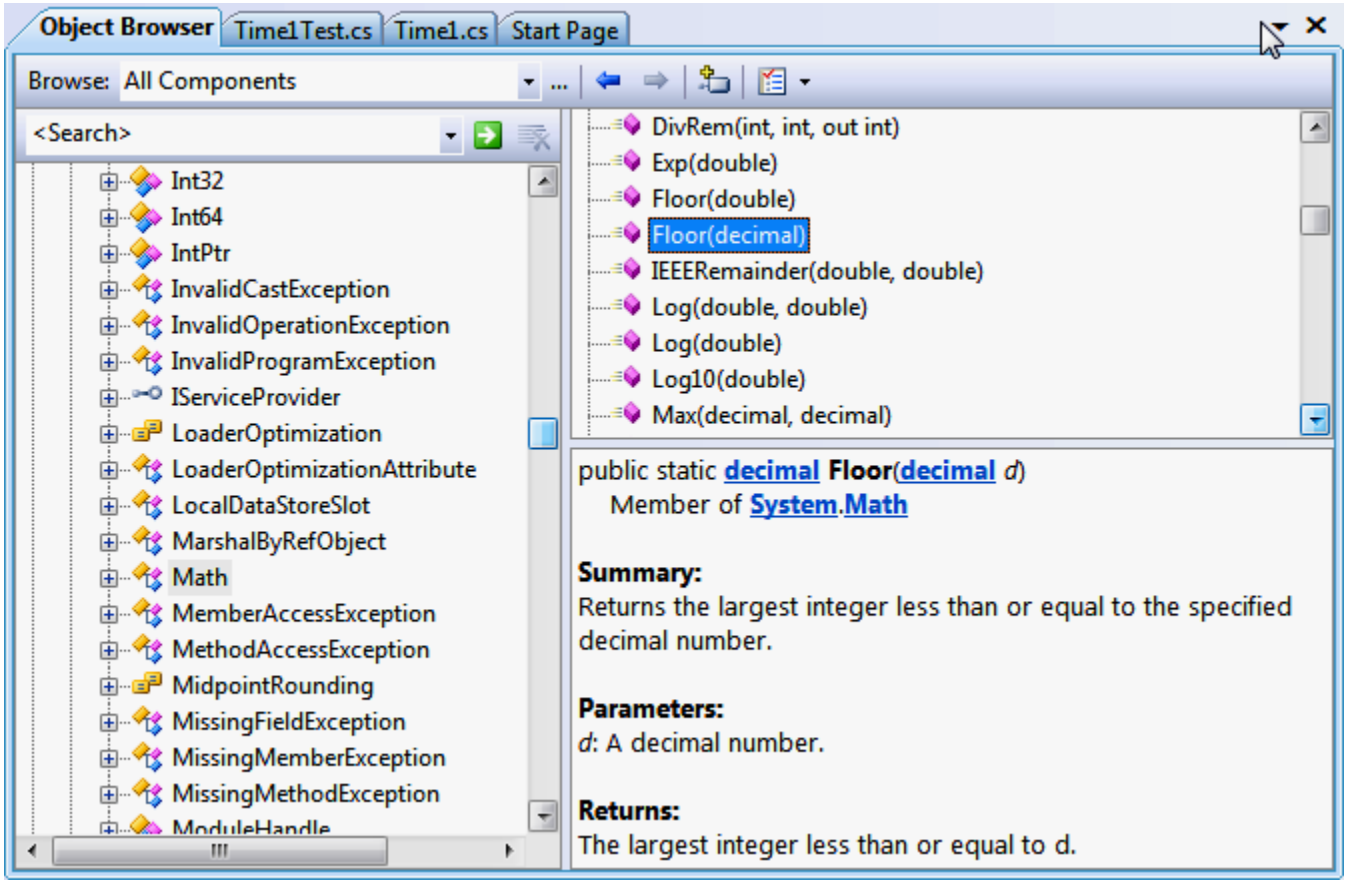

Fig. 10.22 | **Object Browser** for class `Math`.

10.16 Class View and Object Browser (Cont.)

- The **Object Browser** lists all methods provided by class `Math` in the upper-right frame.
- If you click the name of a member in the upper-right frame, a description of that member appears in the lower-right frame.
- The **Object Browser** lists all classes of the Framework Class Library.

- Since 2008 Visual C# provides a new feature—**object initializers**—that allow you to create an object and initialize its properties in the same statement.
- Object initializers are useful when a class does not provide an appropriate constructor to meet your needs.
- For this example, we created a version of the `Time` class (Fig. 10.23) in which we did not define any constructors.

`Time.cs`

(1 of 4)

```
1 // Fig. 10.23: Time.cs
2 // Time class declaration maintains the time in 24-hour format.
3 public class Time
4 {
5    private int hour; // 0 - 23
6    private int minute; // 0 - 59
7    private int second; // 0 - 59
8
```

Fig. 10.23 | `Time` class declaration maintains the time in 24-hour format. (Part 1 of 4.)

Time.cs

(2 of 4)

```
9     // set a new time value using universal time; ensure that
10    // the data remains consistent by setting invalid values to zero
11    public void SetTime( int h, int m, int s )
12    {
13       Hour = h; // validate hour
14       Minute = m; // validate minute
15       Second = s; // validate second
16    } // end method SetTime
17
18    // convert to string in universal-time format (HH:MM:SS)
19    public string ToUniversalString()
20    {
21       return string.Format( "{0:D2}:{1:D2}:{2:D2}",
22          hour, minute, second );
23    } // end method ToUniversalString
24
25    // convert to string in standard-time format (H:MM:SS AM or PM)
26    public override string ToString()
27    {
28       return string.Format( "{0}:{1:D2}:{2:D2} {3}",
29          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
```

Fig. 10.23 | `Time` class declaration maintains the time in 24-hour format. (Part 2 of 4.)

Time.cs

(3 of 4)

```
30          minute, second, ( hour < 12 ? "AM" : "PM" ) );
31    } // end method ToString
32
33    // Properties for getting and setting
34    // property that gets and sets the hour
35    public int Hour
36    {
37       get
38       {
39          return hour;
40       } // end get
41       set
42       {
43          hour = ( ( value >= 0 && value < 24 ) ? value : 0 );
44       } // end set
45    } // end property Hour
46
47    // property that gets and sets the minute
48    public int Minute
49    {
50       get
```

Fig. 10.23 | `Time` class declaration maintains the time in 24-hour format. (Part 3 of 4.)

Time.cs

(4 of 4)

```
51       {
52          return minute;
53       } // end get
54       set
55       {
56          minute = ( ( value >= 0 && value < 60 ) ? value : 0 );
57       } // end set
58    } // end property Minute
59
60    // property that gets and sets the second
61    public int Second
62    {
63       get
64       {
65          return second;
66       } // end get
67       set
68       {
69          second = ( ( value >= 0 && value < 60 ) ? value : 0 );
70       } // end set
71    } // end property Second
72 } // end class Time
```

Fig. 10.23 | `Time` class declaration maintains the time in 24-hour format. (Part 4 of 4.)

- Figure 10.24 demonstrates object initializers.

ObjectInitializerTest.cs

(1 of 2)

```
1  // Fig. 10.24: ObjectInitializerTest.cs
2  // Demonstrate object initializers using class Time.
3  using System;
4
5  class ObjectInitializerTest
6  {
7     static void Main( string[] args )
8     {
9        Console.WriteLine( "Time object created with object initializer" );
10
11       // create a Time object and initialize its properties
12       Time aTime = new Time { Hour = 14, Minute = 145, Second = 12 };
13
14       // display the time in both standard and universal format
15       Console.WriteLine( "Standard time: {0}", aTime.ToString() );
16       Console.WriteLine( "Universal time: {0}\n",
17          aTime.ToUniversalString() );
18
```

The class name is immediately followed by an **object-initializer list**—a comma-separated list in curly braces ({ }) of properties and their values.

Fig. 10.24 | Demonstrate object initializers using class `Time`. (Part 1 of 2.)

ObjectInitializerTest.cs

(2 of 2)

```
19       Console.WriteLine( "Time object created with Minute property set" );
20
21       // create a Time object and initialize its Minute property only
22       Time anotherTime = new Time { Minute = 45 };
23
24       // display the time in both standard and universal format
25       Console.WriteLine( "Standard time: {0}", anotherTime.ToString() );
26       Console.WriteLine( "Universal time: {0}",
27          anotherTime.ToUniversalString() );
28    } // end Main
29 } // end class ObjectInitializerTest
```

```
Time object created with object initializer
Standard time: 2:00:12 PM
Universal time: 14:00:12

Time object created with Minute property set
Standard time: 12:45:00 AM
Universal time: 00:45:00
```

Fig. 10.24 | Demonstrate object initializers using class `Time`. (Part 2 of 2.)

10.17 Object Initializers (Cont.)

- The class name is immediately followed by an **object-initializer list**—a comma-separated list in curly braces ({ }) of properties and their values.
- Each property name can appear only once in the object-initializer list.
- The object-initializer list cannot be empty.
- The object initializer executes the property initializers in the order in which they appear.
- An object initializer first calls the class's constructor, so any values not specified in the object initializer list are given their values by the constructor.

- Since 2008 Visual C#, you can use **extension methods** to add functionality to an existing class without modifying the class's source code.
- Many LINQ capabilities are available as extension methods.
- Figure 10.25 uses extension methods to add functionality to class `Time` (from Section 10.17).

`TimeExtensionsTest.cs`

(1 of 3)

```
1  // Fig. 10.25: TimeExtensionsTest.cs
2  // Demonstrating extension methods.
3  using System;
4
5  class TimeExtensionsTest
6  {
7     static void Main( string[] args )
8     {
9        Time myTime = new Time(); // call Time constructor
10       myTime.SetTime( 11, 34, 15 ); // set the time to 11:34:15
11
```

Fig. 10.25 | Demonstrating extension methods. (Part 1 of 3.)

```
12         // test the DisplayTime extension method
13         Console.Write( "Use the DisplayTime method: " );
14         myTime.DisplayTime();
15
16         // test the AddHours extension method
17         Console.Write( "Add 5 hours to the Time object: " );
18         Time timeAdded = myTime.AddHours( 5 ); // add five hours
19         timeAdded.DisplayTime(); // display the new Time object
20
21         // add hours and display the time in one statement
22         Console.Write( "Add 15 hours to the Time object: " );
23         myTime.AddHours( 15 ).DisplayTime(); // add hours and display time
24
25         // use fully qualified extension-method name to display the time
26         Console.Write( "Use fully qualified extension-method name: " );
27         TimeExtensions.DisplayTime( myTime );
28      } // end Main
29   } // end class TimeExtensionsTest
30
31   // extension-methods class
32   static class TimeExtensions
33   {
34      // display the Time object in console
```

TimeExtensionsTest.cs

(2 of 3)

An extension method is called on an object of the class that it extends as if it were a members of the class. The compiler implicitly passes the object that is used to call the method as the extension method's first argument.

Fig. 10.25 | Demonstrating extension methods. (Part 2 of 3.)

```
35    public static void DisplayTime( this Time aTime )
36    {
37       Console.WriteLine( aTime.ToString() );
38    } // end method DisplayTime
39
40    // add the specified number of hours to the time
41    // and return a new Time object
42    public static Time AddHours( this Time aTime, int hours )
43    {
44       Time newTime = new Time(); // create a new Time object
45       newTime.Minute = aTime.Minute; // set the minutes
46       newTime.Second = aTime.Second; // set the seconds
47
48       // add the specified number of hours to the given time
49       newTime.Hour = ( aTime.Hour + hours ) % 24;
50
51       return newTime; // return the new Time object
52    } // end method AddHours
53 } // end class TimeExtensions
```

```
Use the DisplayTime method: 11:34:15 AM
Add 5 hours to the Time object: 4:34:15 PM
Add 15 hours to the Time object: 2:34:15 AM
Use fully qualified extension-method name: 11:34:15 AM
```

TimeExtensionsTest.cs

(3 of 3)

The `this` keyword before a method's first parameter notifies the compiler that the method extends an existing class.

Fig. 10.25 | Demonstrating extension methods. (Part 3 of 3.)

10.18 Time Class Case Study: Extension Methods (Cont.)

- The `this` keyword before a method's first parameter notifies the compiler that the method extends an existing class.
- An extension method is called on an object of the class that it extends as if it were a members of the class. The compiler implicitly passes the object that is used to call the method as the extension method's first argument.
- The type of an extension method's first parameter specifies the class that is being extended—extension methods must define at least one parameter.
- Extension methods must be defined as `static` methods in a `static` top-level class.

10.18 Time Class Case Study: Extension Methods with Generics

```
// extension methods class
static class TimeExtensions
{
   // display the Time object in console
   public static void DisplayTime<T>( this T aTime )
   {
      Console.WriteLine( aTime.ToString() );
   } // end method DisplayTime

   // add the specified number of hours to the time
   // and return a new Time object
   public static Time AddHours<T> ( this T aTime, int hours ) where T: Time
   {
      Time newTime = new Time(); // create a new Time object
      newTime.Minute = aTime.Minute; // set the minutes
      newTime.Second = aTime.Second; // set the seconds

      // add the specified number of hours to the given time
      newTime.Hour = ( aTime.Hour + hours ) % 24;

      return newTime; // return the new Time object
   } // end method AddHours
} // end class TimeExtensions
```

10.18 Time Class Case Study: Extension Methods with Generics

```
class TimeExtensionsTest
{
   static void Main( string[] args )
   {
      Time myTime = new Time(); // call Time constructor
      myTime.SetTime( 11, 34, 15 ); // set the time to 11:34:15

      // test the DisplayTime extension method
      Console.Write( "Use the DisplayTime method: " );
      myTime.DisplayTime();

      // test the AddHours extension method
      Console.Write( "Add 5 hours to the Time object: " );
      Time timeAdded = myTime.AddHours( 5 ); // add five hours
      timeAdded.DisplayTime(); // display the new Time object

      // add hours and display the time in one statement
      Console.Write( "Add 15 hours to the Time object: " );
      myTime.AddHours( 15 ).DisplayTime(); // add hours and display time

      // use fully qualified extension method name to display the time
      Console.Write( "Use fully qualified extension method name: " );
      TimeExtensions.DisplayTime( myTime );
   } // end Main
} // end class TimeExtensionsTest
```

No changes in the client application are required when the extension methods use generics.

10.18 Time Class Case Study: Extension Methods (Cont.)

- *IntelliSense* displays extension methods with the extended class's instance methods and identifies them with a distinct icon (Fig. 10.26).

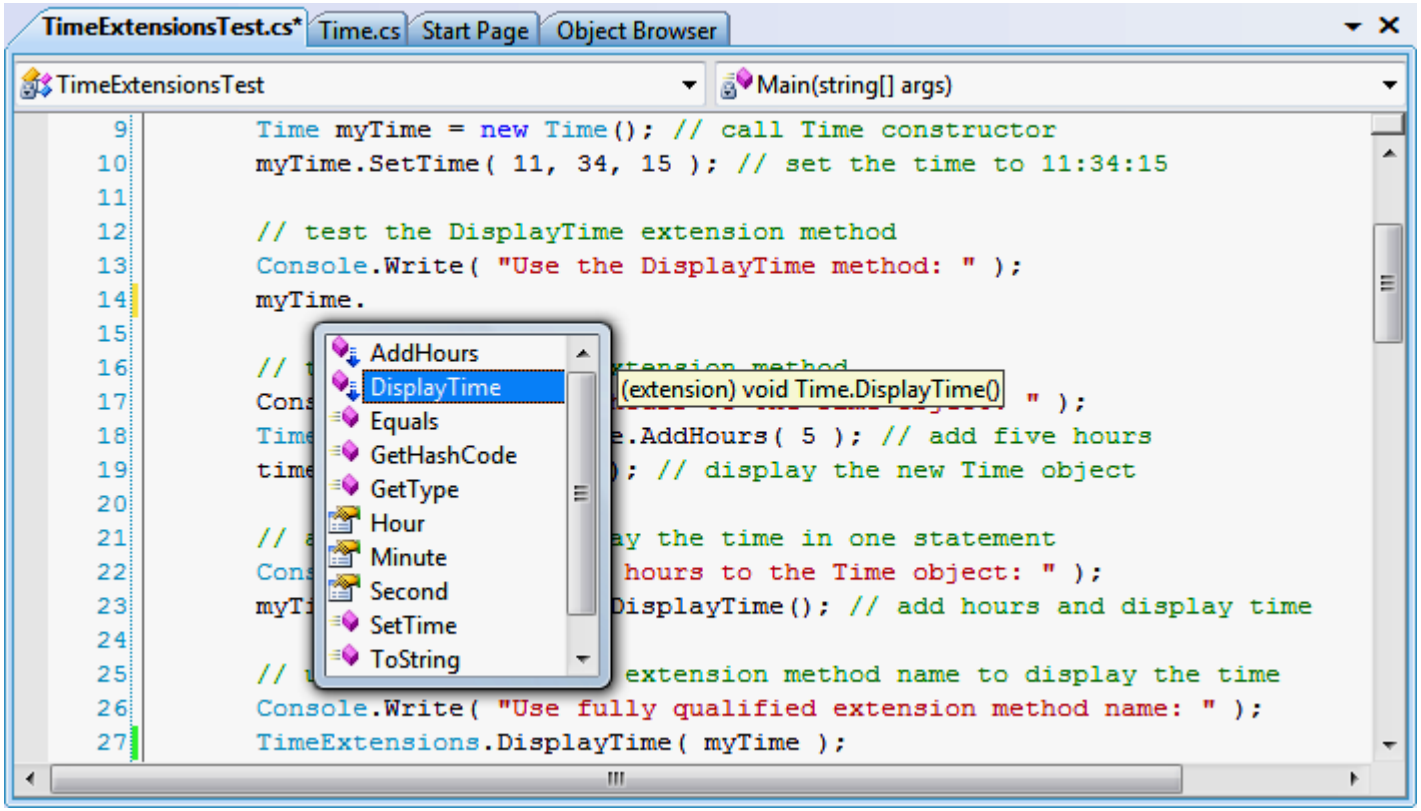

Fig. 10.26 | *IntelliSense* support for extension methods.

10.18 Time Class Case Study: Extension Methods (Cont.)

- Extension methods, as well as instance methods, allow **cascaded method calls**—that is, invoking multiple methods in the same statement.
- Cascaded method calls are performed from left to right.
- When using the fully qualified method name to call an extension method, you must specify an argument for extension method's first parameter. This use of the extension method resembles a call to a `static` method.
- If the type being extended defines an instance method with the **same name as your extension method** and **a compatible signature**, **the instance method will shadow the extension method**.

- A **delegate** is an object that holds a reference to a method.
- Delegates allow you to treat methods as data—via delegates, you can assign methods to variables, and pass methods to and from other methods.
- You can also call methods through variables of delegate types.
- A delegate type is declared by preceeding a method header with keyword `delegate` (placed after any access specifiers, such as `public` or `private`).
- Figure 10.27 uses delegates to customize the functionality of a method that filters an `int` array.

Delegates.cs

(1 of 5)

```
1 // Fig. 10.27: Delegates.cs
2 // Using delegates to pass functions as arguments.
3 using System;
4 using System.Collections.Generic;
5
```

Fig. 10.27 | Using delegates to pass functions as arguments. (Part 1 of 5.)

```
6  class Delegates
7  {
8     // delegate for a function that receives an int and returns a bool
9     public delegate bool NumberPredicate( int number );
10
11    static void Main( string[] args )
12    {
13       int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14
15       // create an instance of the NumberPredicate delegate type
16       NumberPredicate evenPredicate = IsEven;
17
18       // call IsEven using a delegate variable
19       Console.WriteLine( "Call IsEven using a delegate variable: {0}",
20          evenPredicate( 4 ) );
21
22       // filter the even numbers using method IsEven
23       List< int > evenNumbers = FilterArray( numbers, evenPredicate );
24
25       // display the result
26       DisplayList( "Use IsEven to filter even numbers: ", evenNumbers );
27
```

Delegates.cs

(2 of 5)

Define a delegate type named `NumberPredicate`. This variable can store a reference to any method that takes an `int` argument and returns a `bool`.

Because method `IsEven`'s signature matches the `NumberPredicate` delegate's signature, `IsEven` can be referenced by a variable of type `NumberPredicate`.

The method referenced by the delegate is called using the delegate variable's name in place of the method's name.

Fig. 10.27 | Using delegates to pass functions as arguments. (Part 2 of 5.)

```
28       // filter the odd numbers using method IsOdd
29       List< int > oddNumbers = FilterArray( numbers, IsOdd );
30
31       // display the result
32       DisplayList( "Use IsOdd to filter odd numbers: ", oddNumbers );
33
34       // filter numbers greater than 5 using method IsOver5
35       List< int > numbersOver5 = FilterArray( numbers, IsOver5 );
36
37       // display the result
38       DisplayList( "Use IsOver5 to filter numbers over 5: ",
39          numbersOver5 );
40    } // end Main
41
42    // select an array's elements that satisfy the predicate
43    private static List< int > FilterArray( int[] intArray,
44       NumberPredicate predicate )
45    {
46       // hold the selected elements
47       List< int > result = new List< int >();
48
```

Delegates.cs

(3 of 5)

`FilterArray` takes as arguments an `int` array and a `NumberPredicate` that references a method used to filter the array elements.

Fig. 10.27 | Using delegates to pass functions as arguments. (Part 3 of 5.)

Delegates.cs

(4 of 5)

```
49       // iterate over each element in the array
50       foreach ( int item in intArray )
51       {
52          // if the element satisfies the predicate
53          if ( predicate( item ) )
54             result.Add( item ); // add the element to the result
55       } // end foreach
56
57       return result; // return the result
58    } // end method FilterArray
59
60    // determine whether an int is even
61    private static bool IsEven( int number )
62    {
63       return ( number % 2 == 0 );
64    } // end method IsEven
65
66    // determine whether an int is odd
67    private static bool IsOdd( int number )
68    {
69       return ( number % 2 == 1 );
70    } // end method IsOdd
```

Fig. 10.27 | Using delegates to pass functions as arguments. (Part 4 of 5.)

```
71
72    // determine whether an int is positive
73    private static bool IsOver5( int number )
74    {
75       return ( number > 5 );
76    } // end method IsOver5
77
78    // display the elements of a List
79    private static void DisplayList( string description, List< int > list )
80    {
81       Console.Write( description ); // display the output's description
82
83       // iterate over each element in the List
84       foreach ( int item in list )
85          Console.Write( "{0} ", item ); // print item followed by a space
86
87       Console.WriteLine(); // add a new line
88    } // end method DisplayList
89 } // end class Delegates
```

Delegates.cs

(5 of 5)

```
Call IsEven using a delegate variable: True
Use IsEven to filter even numbers: 2 4 6 8 10
Use IsOdd to filter odd numbers: 1 3 5 7 9
Use IsOver5 to filter numbers over 5: 6 7 8 9 10
```

Fig. 10.27 | Using delegates to pass functions as arguments. (Part 5 of 5.)

- **Lambda expressions** allow you to define simple, **anonymous functions**.
- Figure 10.28 uses lambda expressions to reimplement the previous example that introduced delegates.

Lambdas.cs

(1 of 4)

```
1  // Fig. 10.28: Lambdas.cs
2  // Using lambda expressions.
3  using System;
4  using System.Collections.Generic;
5
6  class Lambdas
7  {
8     // delegate for a function that receives an int and returns a bool
9     public delegate bool NumberPredicate( int number );
10
11    static void Main( string[] args )
12    {
13       int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14
15       // create an instance of the NumberPredicate delegate type using an
16       // implicit lambda expression
17       NumberPredicate evenPredicate = number => ( number % 2 == 0 );
```

Equivalent to :
NumberPredicate evenPredicate = **delegate** (int number) { return number % 2 == 0; };.

A **lambda expression** begins with a parameter list, which is followed by the => **lambda operator** and an expression that represents the body of the function.

Fig. 10.28 | Using lambda expressions. (Part 1 of 4.)

```
18
19      // call a lambda expression through a variable
20      Console.WriteLine( "Use a lambda-expression variable: {0}",
21         evenPredicate( 4 ) );
22
23      // filter the even numbers using a lambda expression
24      List< int > evenNumbers = FilterArray( numbers, evenPredicate );
25
26      // display the result
27      DisplayList( "Use a lambda expression to filter even numbers: ",
28         evenNumbers );
29
30      // filter the odd numbers using an explicitly typed lambda
31      // expression
32      List< int > oddNumbers = FilterArray( numbers,
33         ( int number ) => ( number % 2 == 1 ) );
34
35      // display the result
36      DisplayList( "Use a lambda expression to filter odd numbers: ",
37         oddNumbers );
38
```

Lambdas.cs

(2 of 4)

A lambda expression can be called via the variable that references it.

A lambda expression's input parameter `number` can be explicitly typed.

Fig. 10.28 | Using lambda expressions. (Part 2 of 4.)

```
39       // filter numbers greater than 5 using an implicit lambda statement
40       List< int > numbersOver5 = FilterArray( numbers,
41          number => { return number > 5; } );
42
43       // display the result
44       DisplayList( "Use a lambda expression to filter numbers over 5: ",
45          numbersOver5 );
46    } // end Main
47
48    // select an array's elements that satisfy the predicate
49    private static List< int > FilterArray( int[] intArray,
50       NumberPredicate predicate )
51    {
52       // hold the selected elements
53       List< int > result = new List< int >();
54
55       // iterate over each element in the array
56       foreach ( int item in intArray )
57       {
58          // if the element satisfies the predicate
59          if ( predicate( item ) )
```

Lambdas.cs

(3 of 4)

Statement lambdas contain a statement block—a set of statements enclosed in braces ({})—to the right of the lambda operator.

Fig. 10.28 | Using lambda expressions. (Part 3 of 4.)

```
60             result.Add( item ); // add the element to the result
61          } // end foreach
62
63          return result; // return the result
64       } // end method FilterArray
65
66       // display the elements of a List
67       private static void DisplayList( string description, List< int > list )
68       {
69          Console.Write( description ); // display the output's description
70
71          // iterate over each element in the List
72          foreach ( int item in list )
73             Console.Write( "{0} ", item ); // print item followed by a space
74
75          Console.WriteLine(); // add a new line
76       } // end method DisplayList
77    } // end class Lambdas
```

Lambdas.cs

(4 of 4)

```
Use a lambda-expression variable: True
Use a lambda expression to filter even numbers: 2 4 6 8 10
Use a lambda expression to filter odd numbers: 1 3 5 7 9
Use a lambda expression to filter numbers over 5: 6 7 8 9 10
```

Fig. 10.28 | Using lambda expressions. (Part 4 of 4.)

10.20 Lambda Expressions

- A lambda expression begins with a parameter list, which is followed by the => **lambda operator** and an expression that represents the body of the function.
- The value produced by the expression is implicitly returned by the lambda expression.
- The return type can be inferred from the return value or, in some cases, from the delegate's return type.
- A delegate can hold a reference to a **lambda expression** whose signature is compatible with the delegate type.
- Lambda expressions are often **used as arguments to methods** with **parameters of delegate types**, rather than defining and referencing a separate method.

10.20 Lambda Expressions (Cont.)

- A lambda expression can be called via the variable that references it.
- A lambda expression's input parameter `number` can be explicitly typed.
- Lambda expressions that have an expression to the right of the lambda operator are called **expression lambdas**.
- **Statement lambdas** contain a statement block—a set of statements enclosed in braces (`{}`)—to the right of the lambda operator.
- Lambda expressions can help reduce the size of your code and the complexity of working with delegates.
- Lambda expressions are particularly powerful when combined with the `where` clause in LINQ queries.

- **Anonymous types** allow you to create simple classes used to store data without writing a class definition.
- Anonymous type declarations—known formally as **anonymous object-creation expressions**—are demonstrated in Fig. 10.29.

AnonymousTypes.cs

(1 of 3)

```
1  // Fig. 10.29: AnonymousTypes.cs
2  // Using anonymous types.
3  using System;
4
5  class AnonymousTypes
6  {
7     static void Main( string[] args )
8     {
9        // create a "person" object using an anonymous type
10       var bob = new { Name = "Bob Smith", Age = 37 };
11
12       // display Bob's information
13       Console.WriteLine( "Bob: " + bob.ToString() );
14
```

An anonymous type declaration begins with the keyword `new` followed by a member-initializer list in braces ({}).

Fig. 10.29 | Using anonymous types. (Part 1 of 4.)

AnonymousTypes.cs

(2 of 3)

```
15      // create another "person" object using the same anonymous type
16      var steve = new { Name = "Steve Jones", Age = 26 };
17
18      // display Steve's information
19      Console.WriteLine( "Steve: " + steve.ToString() );
20
21      // determine if objects of the same anonymous type are equal
22      Console.WriteLine( "\nBob and Steve are {0}",
23         ( bob.Equals( steve ) ? "equal" : "not equal" ) );
24
25      // create a "person" object using an anonymous type
26      var bob2 = new { Name = "Bob Smith", Age = 37 };
27
28      // display Bob's information
29      Console.WriteLine( "\nBob2: " + bob2.ToString() );
30
31      // determine whether objects of the same anonymous type are equal
32      Console.WriteLine( "\nBob and Bob2 are {0}\n",
```

Because they are anonymous, you must use implicitly typed local variables to reference objects of anonymous types.

The anonymous type's `Equals` method compares the properties of two anonymous objects.

Fig. 10.29 | Using anonymous types. (Part 2 of 3.)

AnonymousTypes.cs

(3 of 3)

```
33          ( bob.Equals( bob2 ) ? "equal" : "not equal" ) );
34    } // end Main
35 } // end class AnonymousTypes
```

```
Bob: { Name = Bob Smith, Age = 37 }
Steve: { Name = Steve Jones, Age = 26 }

Bob and Steve are not equal
Bob2: { Name = Bob Smith, Age = 37 }
Bob and Bob2 are equal
```

Fig. 10.29 | Using anonymous types. (Part 3 of 3.)

10.21 Anonymous Types

- An anonymous type declaration begins with the keyword `new` followed by a member-initializer list in braces (`{}`).
- The compiler generates a new class definition that contains the properties specified in the member-initializer list.
- All properties of an anonymous type are `public` and immutable.
- Anonymous type properties are read-only—you cannot modify a property's value once the object is created.
- Each property's type is inferred from the values assigned to it.
- Because they are anonymous, you must use implicitly typed local variables to reference objects of anonymous types.

10.21 Anonymous Types (Cont.)

- The compiler defines the `ToString` method that returns a `string` in curly braces containing a comma-separated list of *PropertyName* = *value* pairs.
- Two anonymous objects that specify the same property names and types, in the same order, use the same anonymous class definition and are considered to be of the same type.
- The anonymous type's `Equals` method compares the properties of two anonymous objects.

10.21 Anonymous Types (Cont.)

Anonymous Types in LINQ

- Anonymous types are frequently used in LINQ queries to select specific properties from the items being queried.

```
var names =
        from e in employees
        select new { e.FirstName, Last = e.LastName };
```