10

# Classes and Objects: A Deeper Look

# `Time1` Class Declaration

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

`Time1.cs`

(1 of 2 )

```
1  // Fig10.1: 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.)

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

`Time1Test.cs`

(1 of 2 )

```
1  // Fig10.2: 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.)

## 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-imple-mentation details.**

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.
- You can also use keyword `this` *explicitly* in a non-`static` method's body.

ThisTest.cs

(1 of 3 )

```
1  // Fig10.4: 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.)

```
35          "this.ToUniversalString()", this.ToUniversalString(),
36          "ToUniversalString()", ToUniversalString() );
37
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
```

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

ThisTest.cs

(3 of 3 )

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

**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.)

## 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  // Fig10.5: 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  // Fig10.6: 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.)

- **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.

```
// Fig. 10.7: Time2.cs
// Time2 class declaration with overloaded constructors.
public class Time2
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor: initializes each instance variable
   // to zero; ensures that Time2 objects start in a consistent state
   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.)

```
15   // Time2 constructor: hour supplied, minute and second defaulted to 0
16   public Time2( int h ) : this( h, 0, 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 )

```
34      Minute = m; // set the Minute property
35      Second = s; // set the Second property
36   } // end constructor
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.)

```
66     } //end get
67   //property that gets and sets the second
68   public int Second
69   {
70     get
71     {
72       return second;
73     } //end get
```

Time2.cs

(4 of 5 )

**Fig. 10.7** | Time2 class declaration with overloaded constructors. (Part 4 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
```

Time2.cs

(5 of 5 )

**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`).**