10

Classes and Objects: A Deeper Look

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`.

10.7 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.

- 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***.

Date.cs

(1 of 4)

Software Engineering Observation 10.6

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

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

```
1  // Fig10.9: 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)

```
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)

```
62
63        {
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  // Fig10.10: 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.

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

`EmployeeTest.cs`

```
1  // Fig10.11: 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.
  - 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

- 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.)

- `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.

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 mod-ify 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.

- 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)

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  // Fig10.16: 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.

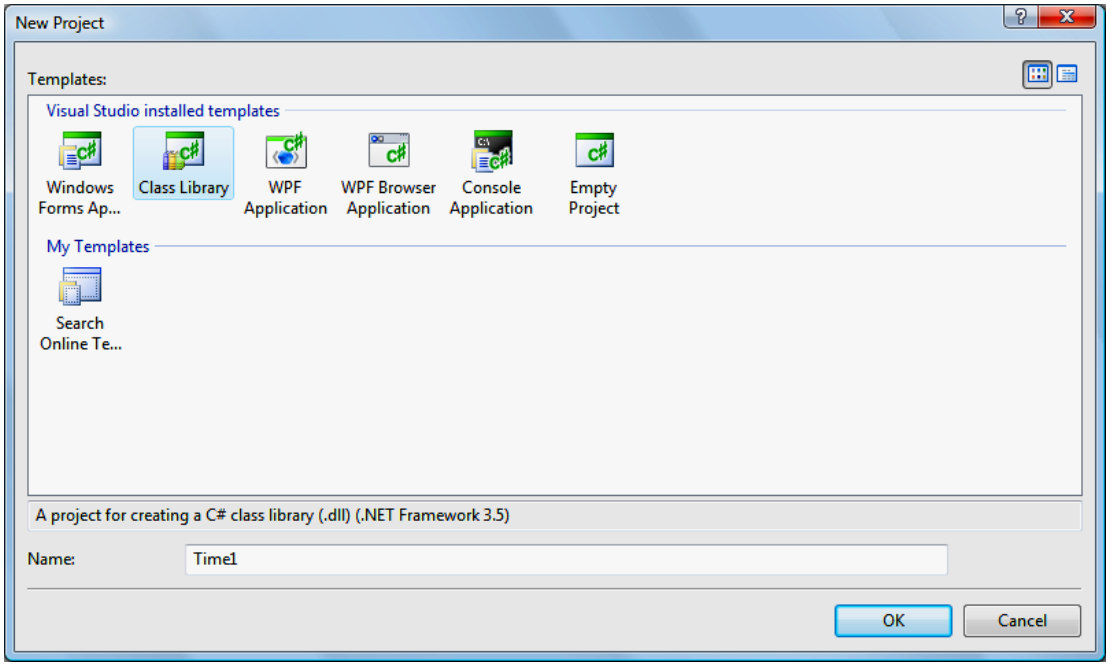

Fig. 10.17 | Creating a Class Library Project.

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.)

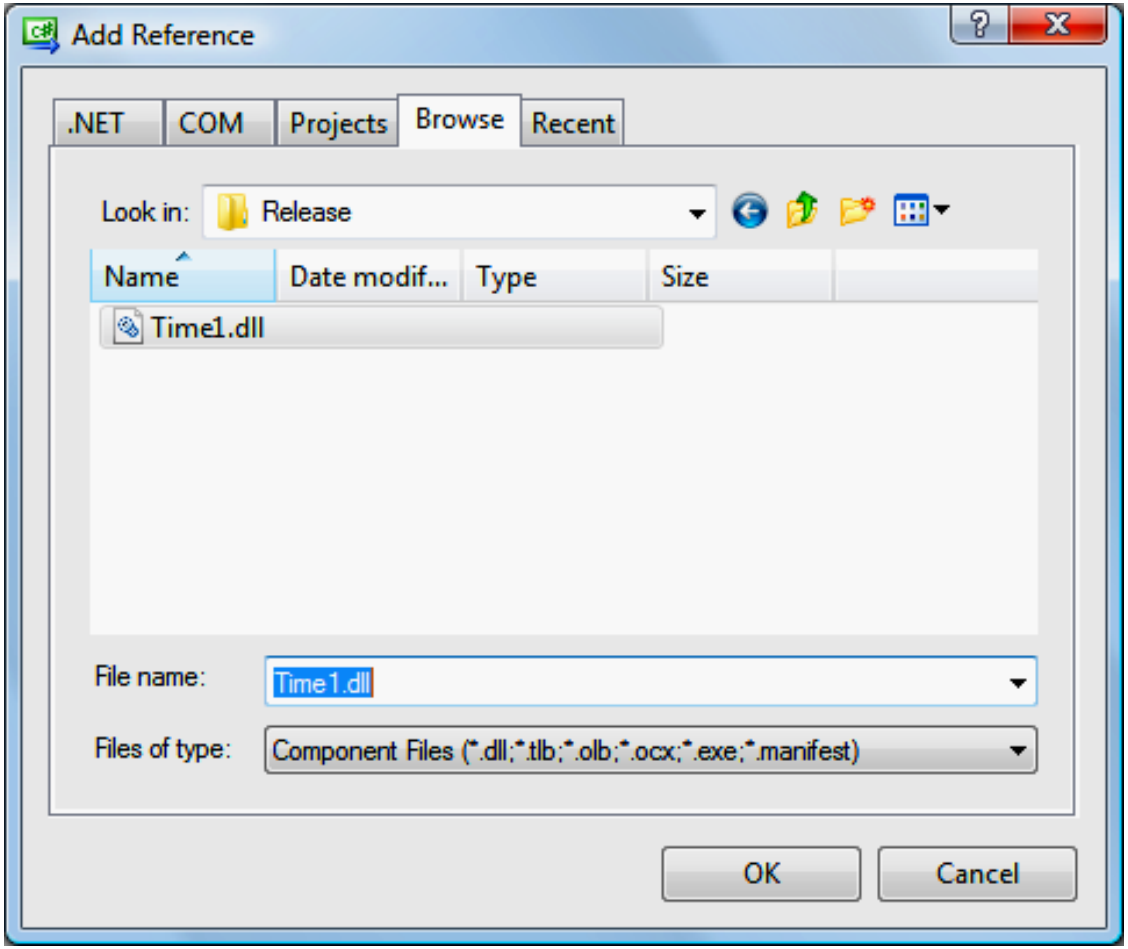

Fig. 10.18 | Adding a Reference.

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).

`Time1Namespace Test.cs`

(1 of 3)

```
1  // Fig10.19: 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.)

Time1Namespace Test.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  // Fig10.20: 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.)

```
19        // output string representation of internalData
20        Console.WriteLine( "After 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()
```

InternalAccessTest.cs

(2 of 3)

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.)

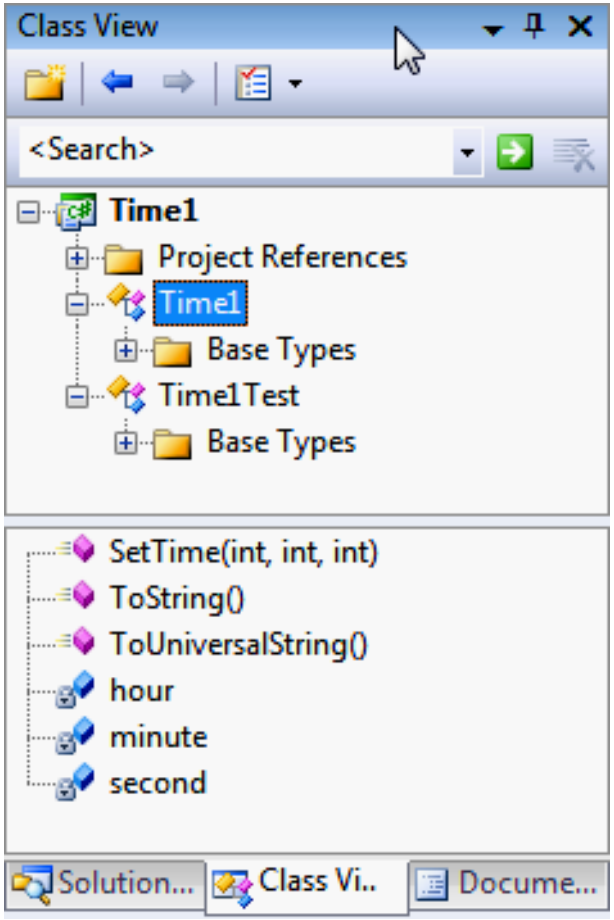

Fig. 10.21 | Class View of class `Time1` (Fig. 10.1) and class `TimeTest` (Fig. 10.2).

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.)

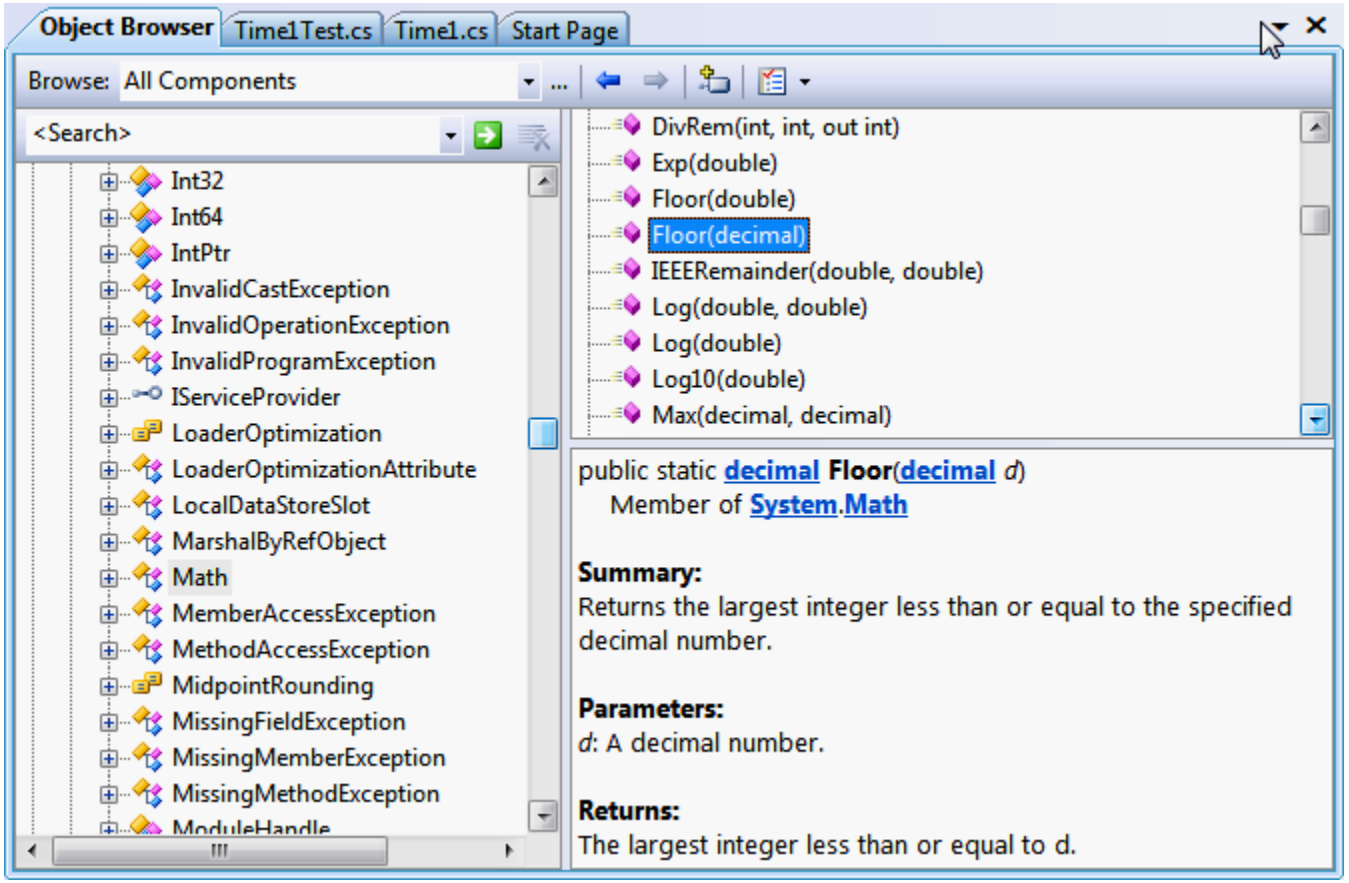

Fig. 10.22 | Object Browser for class `Math`.

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.

- Visual C# 2008 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  // Fig10.23: 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.)

```
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
```

Time.cs

(3 of 4)

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  // Fig10.24: 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.)

```
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
```

ObjectInitializerTest.cs

(2 of 2)

```
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.