Classes and Objects :

A deeper look

Source : [BOOK] How to programming C#

Introduction

In this chapter, we take a deeper look at **building classes**, **controlling access to members** of a class and creating **constructors**. We discuss **composition**—a capability that allows a class to have **references to objects of other classes** as members. The chapter also discusses **static** class members and **read-only** instance variables in **detail**. We also discuss several miscellaneous **topics related to defining classes**.

Time1 Class Case Study

Class Time1 represents **the time of day**. Class ***Time1Test*'s Main method** creates an object of class *Time1* and invokes its methods.

Class *Time1* contains three private instance variables: **hour, minute and second**—that represent the time in universal-time format (24-hour clock format, in which hours are in the range 0–23). Class *Time1* contains public methods **SetTime**, **ToUniversalString** and **ToString**. These are the **public services** or the **public** interface that this class provides to its clients

```
// Fig. 10.1: Time1.cs
// Time1 class declaration maintains the time in 24-hour format.
using System; // namespace containing ArgumentOutOfRangeException

public class Time1
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void SetTime( int h, int m, int s )
   {
      // validate hour, minute and second
      if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
         ( s >= 0 && s < 60 ) )
      {
         hour = h;
         minute = m;
         second = s;
      } // end if
      else
         throw new ArgumentOutOfRangeException();
   } // end method SetTime
// convert to string in universal-time format (HH:MM:SS)
   public string ToUniversalString()
   {
      return string.Format( "{0:D2}:{1:D2}:{2:D2}",
         hour, minute, second );
   } // end method ToUniversalString

   // convert to string in standard-time format (H:MM:SS AM or PM)
   public override string ToString()
   {
      return string.Format( "{0}:{1:D2}:{2:D2} {3}",
         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
         minute, second, ( hour < 12 ? "AM" : "PM" ) );
   } // end method ToString
} // end class Time1
```

Method SetTime and Throwing Exceptions

The class **has a default constructor** that's supplied by the compiler. Each instance variable implicitly receives the **default value 0** for an int.

Method ***SetTime*** is a public method that declares three int parameters and uses them to set the time.

Test each argument to determine whether the **value is in the proper range**.

Method SetTime and Throwing Exceptions

Continue

```
throw new ArgumentOutOfRangeException()
```

For values outside these ranges, ***SetTime*** throws an exception of type ***ArgumentOutOfRangeException***, which notifies the **client code** that an **invalid argument** was passed to the method.

The throw statement creates a new object of type ***ArgumentOutOfRangeException***.

The parentheses following the class name indicate a call to the ***ArgumentOutOfRangeException*** constructor.

After the exception object is created, the throw statement **immediately terminates** method ***SetTime*** and the exception is returned to the code that attempted to set the time.

Method *ToUniversalString*

```
public string ToUniversalString()
  {
     return string.Format( "{0:D2}:{1:D2}:{2:D2}",
        hour, minute, second );
  }
```

Takes no arguments and **returns a string in universal-time format**, for example **13:30:07**.

Uses static method **Format of class string** to return a string containing the formatted hour, minute and second values.

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.

Method *ToString*

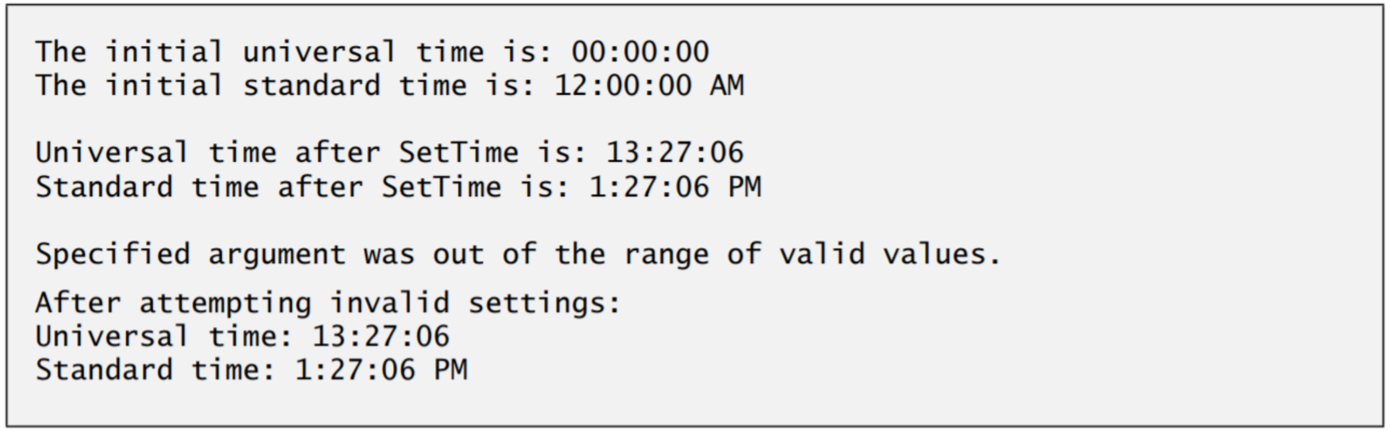
```
public override string ToString()
  {
     return string.Format( "{0}:{1:D2}:{2:D2} {3}",
        ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
        minute, second, ( hour < 12 ? "AM" : "PM" ) );
  }
```

Takes no arguments and returns a string in ***standard time* format** (e.g., 1:27:06 PM).

Using conditional operator (?:) to determine the **value for hour in the string**—if the hour is 0 or 12 (AM or PM), it appears as 12—otherwise, it appears as a value from 1 to 11.

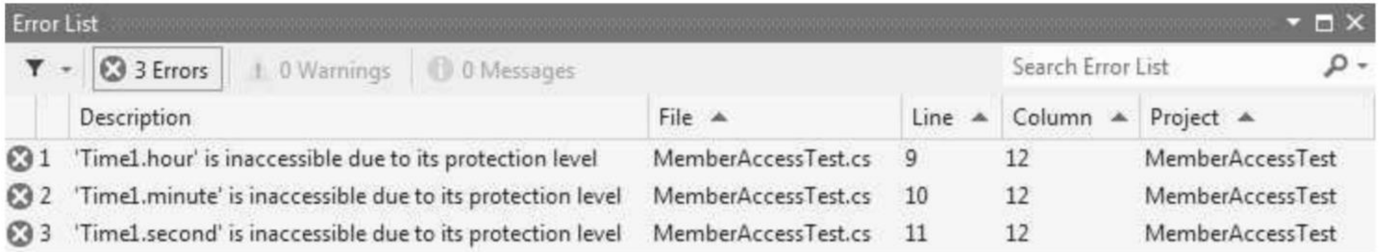
The second conditional operator determines whether AM or PM will be returned as part of the string.

```
// Fig. 10.2: Time1Test.cs
// Time1 object used in an application.
using System;

public class Time1Test
{
   public static void Main( string[] args )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      Console.Write( "The initial universal time is: " );
      Console.WriteLine( time.ToUniversalString() );
      Console.Write( "The initial standard time is: " );
      Console.WriteLine( time.ToString() );
      Console.WriteLine(); // output a blank line

      // change time and output updated time
      time.SetTime( 13, 27, 6 );
      Console.Write( "Universal time after SetTime is: " );
      Console.WriteLine( time.ToUniversalString() );
      Console.Write( "Standard time after SetTime is: " );
      Console.WriteLine( time.ToString() );
      Console.WriteLine(); // output a blank line
// attempt to set time with invalid values
      try
      {
         time.SetTime( 99, 99, 99 );
      } // end try
      catch ( ArgumentOutOfRangeException ex )
      {
         Console.WriteLine( ex.Message + "\n" );
      } // end catch

      // display time after attempt to set invalid values
      Console.WriteLine( "After attempting invalid settings:" );
      Console.Write( "Universal time: " );
      Console.WriteLine( time.ToUniversalString() );
      Console.Write( "Standard time: " );
      Console.WriteLine( time.ToString() );
   } // end Main
} // end class Time1Test
```



```
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

Specified argument was out of the range of valid values.

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

```
// Fig. 10.3: MemberAccessTest.cs
// Private members of class Time1 are not accessible outside the class.
public class MemberAccessTest
{
   public static void Main( string[] args )
   {
      Time1 time = new Time1(); // create and initialize Time1 object

      time.hour = 7; // error: hour has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
      time.second = 30; // error: second has private access in Time1
   } // end Main
} // end class MemberAccessTest
```



	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

Notice that members of a class—for instance, properties, methods and instance variables—do not need to be explicitly declared private. If a class member is not declared with an access modifier, it has **private access by default**.

Controlling Access to Members

Clients of the class need not be concerned with **how** the class accomplishes its tasks.

For this reason, a class’s private variables, properties and methods (i.e., the class’s implementation details) are **not directly accessible** to the class’s clients => **these private members are not accessible**.

Referring to the Current Object’s Members with the this Reference

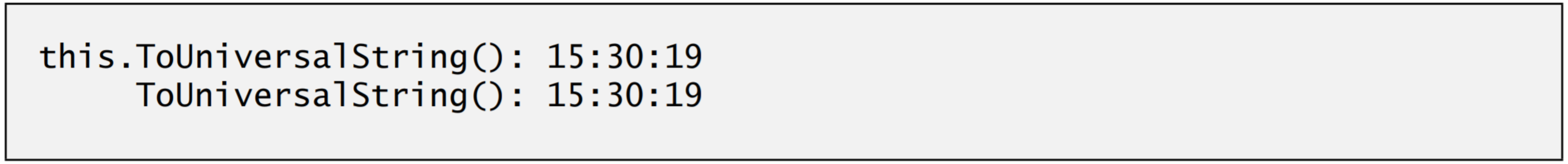
Every object can access a **reference to itself with keyword this**.

When a non-static method is called for a particular object, 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**.

Keyword this **cannot be used in a static method**.

```
// Fig. 10.4: ThisTest.cs
// this used implicitly and explicitly to refer to members of an object.
using System;

public class ThisTest
{
   public static void Main( string[] args )
   {
      SimpleTime time = new SimpleTime( 15, 30, 19 );
      Console.WriteLine( time.BuildString() );
   } // end Main
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
public class SimpleTime
{
   private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59

   // if the constructor uses parameter names identical to
   // instance variable names the "this" reference is
   // required to distinguish between names
   public SimpleTime( int hour, int minute, int second )
   {
      this.hour = hour; // set "this" object's hour instance variable
      this.minute = minute; // set "this" object's minute
      this.second = second; // set "this" object's second
   } // end SimpleTime constructor
// use explicit and implicit "this" to call ToUniversalString
   public string BuildString()
   {
      return string.Format( "{0,24}: {1}\n{2,24}: {3}",
         "this.ToUniversalString()", this.ToUniversalString(),
         "ToUniversalString()", ToUniversalString() );
   } // end method BuildString

   // convert to string in universal-time format (HH:MM:SS)
   public string ToUniversalString()
   {
      // "this" is not required here to access instance variables,
      // because method does not have local variables with same
      // names as instance variables
      return string.Format( "{0:D2}:{1:D2}:{2:D2}",
         this.hour, this.minute, this.second );
   } // end method ToUniversalString
} // end class SimpleTime
```



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

Time2 Class Case Study: Overloaded Constructors

Class Time2 with Overloaded Constructors

Class Time1 **doesn't enable** the class's clients to initialize the time with specific nonzero values.

Class Time2 contains **overloaded constructors**.

One constructor **invokes the other constructor**, which in turn **calls SetTime** to set the hour, minute and second.

The compiler **invokes the appropriate Time2 constructor by matching the number and types of the argument.**

```
// Fig. 10.5: Time2.cs
// Time2 class declaration with overloaded constructors.
using System; // for class ArgumentOutOfRangeException

public class Time2
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // constructor can be called with zero, one, two or three arguments
   public Time2( int h = 0, int m = 0, int s = 0 )
   {
      SetTime( h, m, s ); // invoke SetTime to validate time
   } // end Time2 three-argument constructor

   // Time2 constructor: another Time2 object supplied as an argument
   public Time2( Time2 time )
      : this( time.Hour, time.Minute, time.Second ) { }

   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void SetTime( int h, int m, int s )
   {
      Hour = h; // set the Hour property
      Minute = m; // set the Minute property
      Second = s; // set the Second property
   } // end method SetTime
```

```
// property that gets and sets the hour
   public int Hour
   {
      get
      {
         return hour;
      } // end get
      set
      {
         if ( value >= 0 && value < 24 )
            hour = value;
         else
            throw new ArgumentOutOfRangeException(
               "Hour", value, "Hour must be 0-23" );
      } // end set
   } // end property Hour
// property that gets and sets the minute
   public int Minute
   {
      get
      {
         return minute;
      } // end get
      set
      {
         if ( value >= 0 && value < 60 )
            minute = value;
         else
            throw new ArgumentOutOfRangeException(
               "Minute", value, "Minute must be 0-59" );
      } // end set
   } // end property Minute
```

```
// property that gets and sets the second
   public int Second
   {
      get
      {
         return second;
      } // end get
      set
      {
         if ( value >= 0 && value < 60 )
            second = value;
         else
            throw new ArgumentOutOfRangeException(
               "Second", value, "Second must be 0-59" );
      } // end set
   } // end property Second
```

```csharp
// convert to string in universal-time format (HH:MM:SS)
   public string ToUniversalString()
   {
      return string.Format(
         "{0:D2}:{1:D2}:{2:D2}", Hour, Minute, Second );
   } // end method ToUniversalString

   // convert to string in standard-time format (H:MM:SS AM or PM)
   public override string ToString()
   {
      return string.Format( "{0}:{1:D2}:{2:D2} {3}",
         ( ( Hour == 0 || Hour == 12 ) ? 12 : Hour % 12 ),
         Minute, Second, ( Hour < 12 ? "AM" : "PM" ) );
   } // end method ToString
} // end class Time2
```

Class Time2’s Parameter-less Constructor

```
public Time2( int h = 0, int m = 0, int s = 0 )
{
    SetTime( h, m, s ); // invoke SetTime to validate time
}
```

Can be considered as the class’s **parameter-less constructor**.

Can also be called with **one** argument for the hour, **two** arguments for the hour and minute, or **three** arguments for the hour, minute and second.

calls SetTime to set the time.

Class Time2's Constructor That Receives a Reference to Another Time2 Object

```
public Time2( Time2 time )
      : this( time.Hour, time.Minute, time.Second ) { }
```

A constructor that **receives a reference to a Time2** object.

The values from the argument are **passed to the three-parameter constructor** to initialize the hour, minute and second.

In this constructor, we use **this** in a manner that's allowed only in the **constructor's header**. The usual constructor header is followed by a colon **(:),** then the keyword **this**. The this reference is used in method-call syntax (along with the three int arguments) to **invoke the Time2 constructor that takes three int arguments**.

Constructor Initializers

Constructor initializers are a popular way to **reuse initialization code provided by one of the class's constructors** rather than defining similar code in another constructor's body.

This syntax makes the class **easier to maintain**, because one constructor reuses the other. If we needed to change how objects of class Time2 are initialized, only the first **constructor would need to be modified**.

```
public Time2(int minute, int second)
  : this(0, minute, second) { }
```

Class Time2’s properties

```
 get
{
         return hour;
}
 set
{
         if ( value >= 0 && value < 24 )
            hour = value;
         else
            throw new ArgumentOutOfRangeException(
               "Hour", value, "Hour must be 0-23" );
}
```

Hour, Minute and Second properties ensure that the value supplied for hour is in the range 0 to 23 and that the values for minute and second are each in the range 0 to 59.

If a value is **out of range, each set *accessor* throws an *ArgumentOutOfRangeException***.

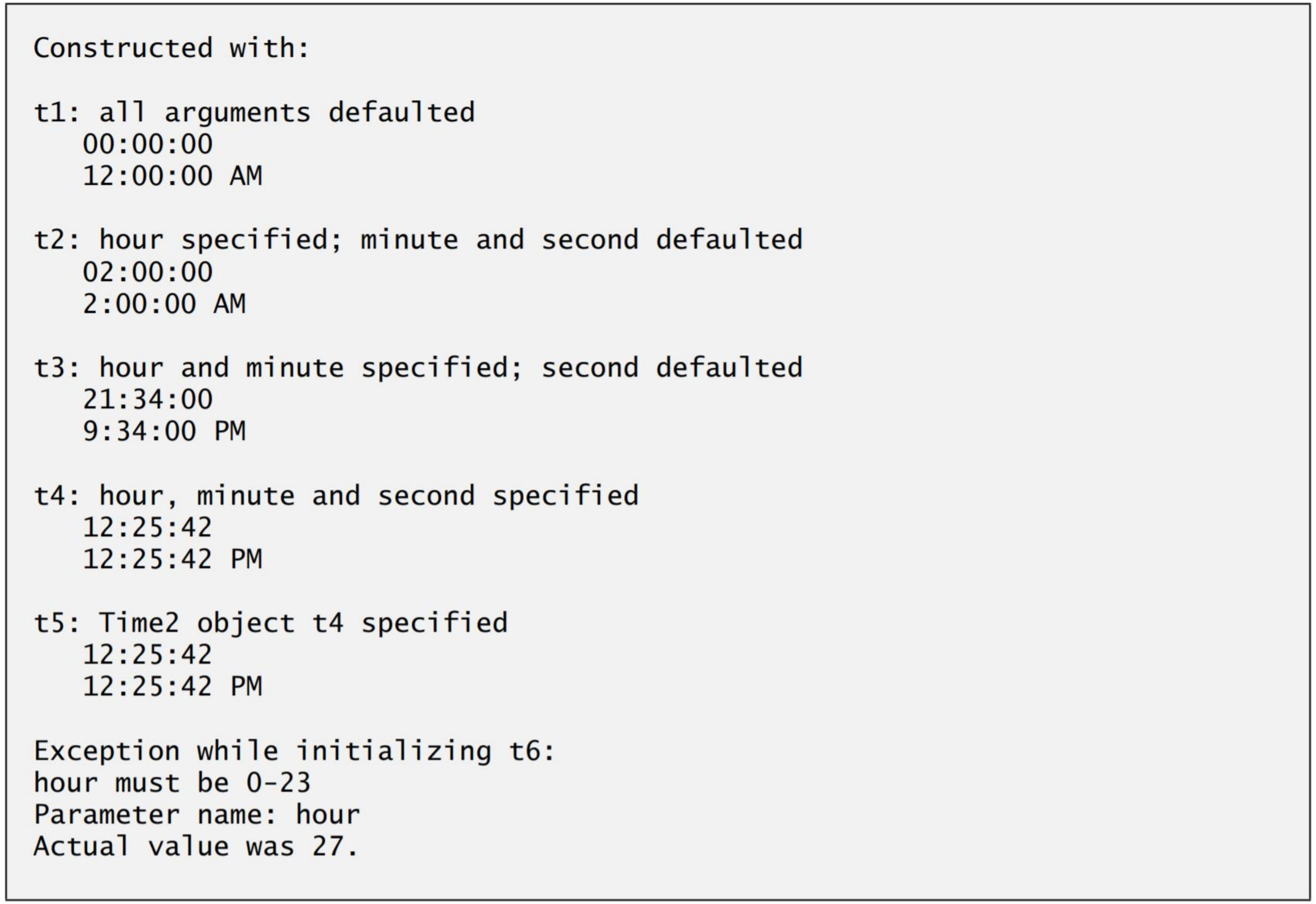
Used *ArgumentOutOfRangeException* constructor that receives **three arguments**—the name of the item that was out of range, the value that was supplied for that item and an error message.

Software Engineering Observation

When implementing a method of a class, **using the class’s properties** to access the class’s private data simplifies **code maintenance** and reduces the **likelihood of errors**.

```
// Fig. 10.6: Time2Test.cs
// Overloaded constructors used to initialize Time2 objects.
using System;

public class Time2Test
{
   public static void Main( string[] args )
   {
      Time2 t1 = new Time2(); // 00:00:00
      Time2 t2 = new Time2( 2 ); // 02:00:00
      Time2 t3 = new Time2( 21, 34 ); // 21:34:00
      Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
      Time2 t5 = new Time2( t4 ); // 12:25:42
      Time2 t6; // initialized later in the program
      Console.WriteLine( "Constructed with:\n" );
      Console.WriteLine( "t1: all arguments defaulted" );
      Console.WriteLine( "   {0}", t1.ToUniversalString() ); // 00:00:00
      Console.WriteLine( "   {0}\n", t1.ToString() ); // 12:00:00 AM

      Console.WriteLine( "t2: hour specified; minute and second defaulted" );
      Console.WriteLine( "   {0}", t2.ToUniversalString() ); // 02:00:00
      Console.WriteLine( "   {0}\n", t2.ToString() ); // 2:00:00 AM

      Console.WriteLine( "t3: hour and minute specified; second defaulted" );
      Console.WriteLine( "   {0}", t3.ToUniversalString() ); // 21:34:00
      Console.WriteLine( "   {0}\n", t3.ToString() ); // 9:34:00 PM

      Console.WriteLine( "t4: hour, minute and second specified" );
      Console.WriteLine( "   {0}", t4.ToUniversalString() ); // 12:25:42
      Console.WriteLine( "   {0}\n", t4.ToString() ); // 12:25:42 PM

      Console.WriteLine( "t5: Time2 object t4 specified" );
      Console.WriteLine( "   {0}", t5.ToUniversalString() ); // 12:25:42
      Console.WriteLine( "   {0}", t5.ToString() ); // 12:25:42 PM
      // attempt to initialize t6 with invalid values
      try
      {
         t6 = new Time2( 27, 74, 99 ); // invalid values
      } // end try
      catch ( ArgumentOutOfRangeException ex )
      {
         Console.WriteLine( "\nException while initializing t6:" );
         Console.WriteLine( ex.Message );
      } // end catch
   } // end Main
} // end class Time2Test
```

Result



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

Exception while initializing t6:
hour must be 0-23
Parameter name: hour
Actual value was 27.
```

Default and Parameter less Constructors

```
public Time2( int h = 0, int m = 0, int s = 0 )
  {
     SetTime( h, m, s ); // invoke SetTime to validate time
  }
```

Like a default constructor, a *parameter less* constructor is invoked with **empty parentheses**.

The compiler will not create a default constructor for a class that explicitly declares at least one constructor.

Software Engineering Observation

One form of software reuse is **composition**, in which a class **contains references to other objects**.

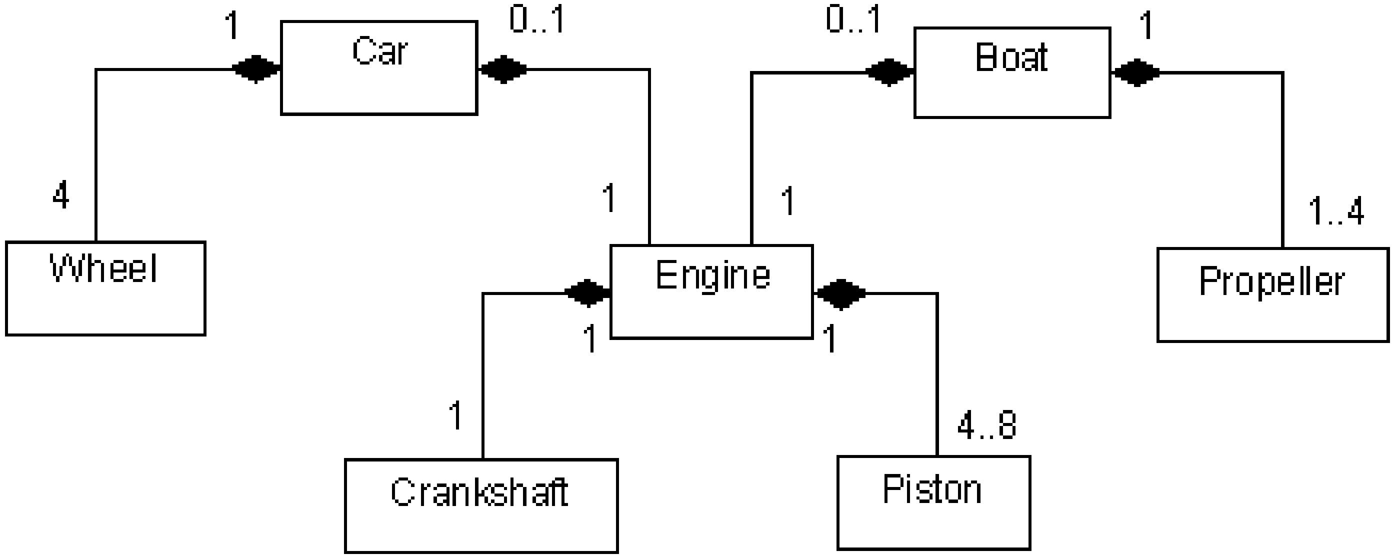
A class can have a property of its own type—for example, a Person class could have a Mom property of type Person.

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**.

For example, an object of class ***AlarmClock*** needs to know the current time and the time when it’s supposed to sound its alarm, so it’s reasonable to include ***two* references to Time** objects in an *AlarmClock* object


Car
Wheel
Engine
Boat
Propeller
Crankshaft
Piston
1
4
0..1
1
0..1
1
1
1..4
1
1
1
1
4..8

Class Date

Our example of composition contains three classes—**Date** (Fig. 10.7), **Employee** (Fig. 10.8) and ***EmployeeTest*** (Fig. 10.9).

```
// Fig. 10.7: Date.cs
// Date class declaration.
using System;

public class Date
{
   private int month; // 1-12
   private int day; // 1-31 based on month

   // auto-implemented property Year
   public int Year { get; private set; }

   // constructor: use property Month to confirm proper value for
month;
   // use property Day to confirm proper value for day
   public Date( int theMonth, int theDay, int theYear )
   {
      Month = theMonth; // validate month
      Year = theYear; // could validate year
      Day = theDay; // validate day
      Console.WriteLine( "Date object constructor for date {0}",
this );
   } // end Date constructor
```

```
// property that gets and sets the month
   public int Month
   {
      get
      {
         return month;
      } // end get
      private set // make writing inaccessible outside the class
      {
         if ( value > 0 && value <= 12 ) // validate month
            month = value;
         else // month is invalid
            throw new ArgumentOutOfRangeException(
               "Month", value, "Month must be 1-12" );
      } // end set
   } // end property Month
```

```
// property that gets and sets the day
   public int Day
   {
      get
      {
         return day;
      } // end get
      private set // make writing inaccessible outside the class
      {
         int[] daysPerMonth = { 0, 31, 28, 31, 30, 31, 30,
                                 31, 31, 30, 31, 30, 31 };
         // check if day in range for month
         if ( value > 0 && value <= daysPerMonth[ Month ] )
            day = value;
         // check for leap year
         else if ( Month == 2 && value == 29 &&
            ( Year % 400 == 0 || ( Year % 4 == 0 && Year % 100 != 0 ) ) )
            day = value;
         else // day is invalid
            throw new ArgumentOutOfRangeException(
               "Day", value, "Day out of range for current month/year" );
      } // end set
   } // end property Day
   // return a string of the form month/day/year
   public override string ToString()
   {
      return string.Format( "{0}/{1}/{2}", Month, Day, Year );
   } // end method ToString
} // end class Date
```

Class Date

Continue

The **order of initialization** is important, because the set accessor of property Day validates the value for day based on the assumption that month and Year are correct.

The day setter determines whether the day is correct based on the number of days in the particular Month.

```
Console.WriteLine( "Date object constructor for date {0}", this );
```

Outputs the this reference as a string. Since this is a reference to the current Date object, the object’s ToString method is called **implicitly** to obtain the object’s string representation.

```
// Fig. 10.8: Employee.cs
// Employee class with references to other objects.
public class Employee
{
   public string FirstName { get; private set; }
   public string LastName { get; private set; }
   public Date BirthDate { get; private set; }
   public Date HireDate { get; private set; }

   // constructor to initialize name, birth date and hire date
   public Employee( string first, string last,
      Date dateOfBirth, Date dateOfHire )
   {
      FirstName = first;
      LastName = last;
      BirthDate = dateOfBirth;
      HireDate = dateOfHire;
   } // end Employee constructor

   // convert Employee to string format
   public override string ToString()
   {
      return string.Format( "{0}, {1}  Hired: {2}  Birthday: {3}",
         LastName, FirstName, HireDate, BirthDate );
   } // end method ToString
} // end class Employee
```

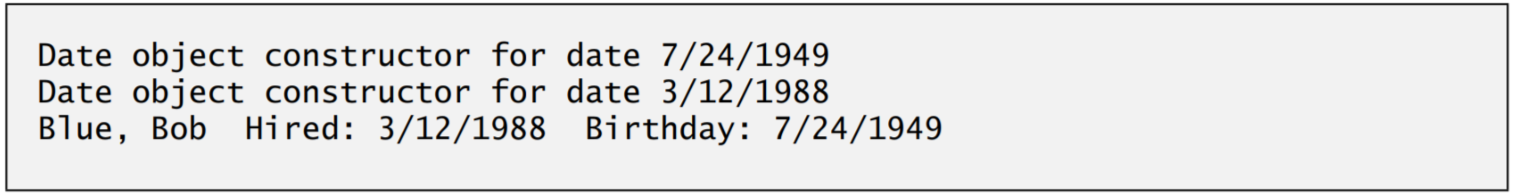
Class Employee

Class Employee has **public auto-implemented properties** *FirstName*, *LastName*, *BirthDate* and *HireDate*.

BirthDate and *HireDate* manipulate Date objects, demonstrating that a class can **have references to objects of other classes** as members.

This, of course, is also true of the properties *FirstName* and *LastName*, which manipulate **String objects**.

```
// Fig. 10.9: EmployeeTest.cs
// Composition demonstration.
using System;

public class EmployeeTest
{
   public static void Main( string[] args )
   {
      Date birth = new Date( 7, 24, 1949 );
      Date hire = new Date( 3, 12, 1988 );
      Employee employee = new Employee( "Bob", "Blue",
birth, hire );

      Console.WriteLine( employee );
   } // end Main
} // end class EmployeeTest
```



```
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
```

Garbage Collection and Destructors

Every object you create **uses various system resources, such as memory**.

Should **explicitly released** in some programming languages.

If **all the references to the object that manages the resource are lost** before the resource is explicitly released, the app can no longer access the resource to release it. This is known as a **resource leak**.

avoiding resource leaks => disciplined way to **give resources back to the system** when they’re no longer needed => CLR performs **automatic memory management** by using a **garbage collector** to reclaim the memory occupied by objects => The memory can be used for other objects.

When there are **no more references to an object**, the object becomes eligible for destruction.

Garbage Collection and Destructors

Continue

Every object has a **destructor**

Destructor **invoked by the garbage collector**

Declared like a **parameter less constructo**r, except that its name is the class name, preceded by a tilde (**~**), and it has **no access modifier** in its header.

After the garbage collector calls the object's destructor, the object becomes eligible for garbage collection.

=> Memory leaks are **less likely in C#**

Garbage Collection and Destructors

Continue

Other types of resource leaks: an app could **open a file** on disk to modify its contents. If the app does not **close the file**, no other app can modify (or possibly even use) the file until the app that opened it terminates.

A problem with the garbage collector is that **it doesn’t guarantee that it will perform its tasks at a specified time**. Therefore, the **garbage collector may call the destructor any time** after the object becomes eligible for destruction => des**tructors are rarely used**.

Software Engineering Observation

A class that uses resources, such as files on disk, **should provide a method to eventually release the resources**

Close or Dispose methods in .net Framework Class Library

static Class Members

Suppose that we have a **video game** with Martians and other space creatures. Each Martian tends to be brave and willing to attack other space creatures when it’s aware that there are at least four other Martians present. **If fewer than five Martians are present**, each Martian becomes **cowardly**. Thus each **Martian needs to know the martianCount**.

martianCount as an instance variable **wastes space on redundant copies and wastes time updating**

Static martinCount => saves **space** & **save time** by having the **Martian constructor increment the static martianCoun**t

Software Engineering Observation

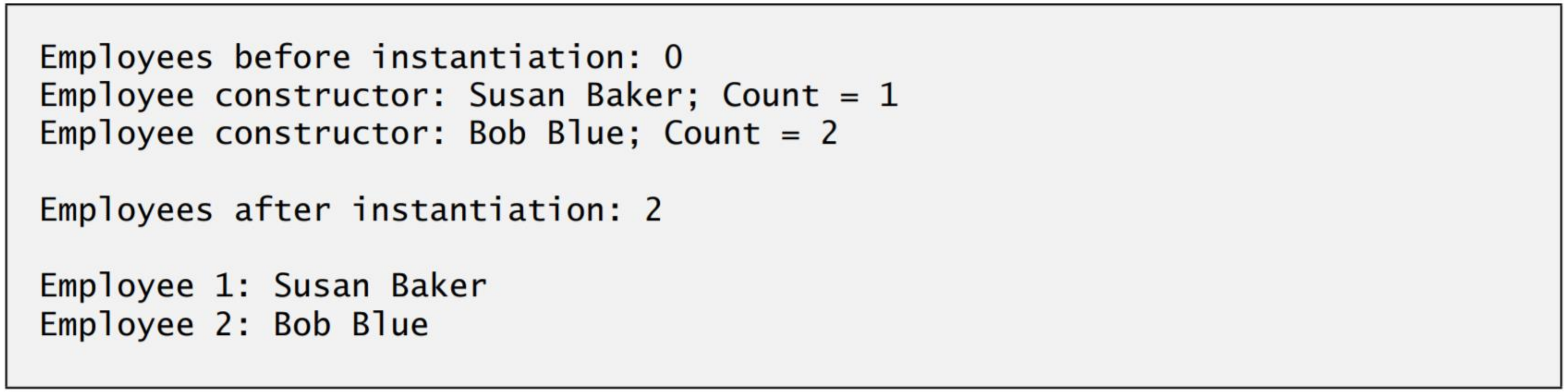
Use a static variable when all objects of a class **must share the same copy of the variable**.

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

```
// Fig. 10.10: Employee.cs
// Static variable used to maintain a count of the number of
// Employee objects that have been created.
using System;

public class Employee
{
   public static int Count { get; private set; } // objects in memory

   // read-only auto-implemented property FirstName
   public string FirstName { get; private set; }

   // read-only auto-implemented property LastName
   public string LastName { get; private set; }

   // initialize employee, add 1 to static Count and
   // output string indicating that constructor was called
   public Employee( string first, string last )
   {
      FirstName = first;
      LastName = last;
      ++Count; // increment static count of employees
      Console.WriteLine( "Employee constructor: {0} {1}; Count = {2}",
         FirstName, LastName, Count );
   } // end Employee constructor
} // end class Employee
```

```
// Fig. 10.11: EmployeeTest.cs
// Static member demonstration.
using System;

public class EmployeeTest
{
   public static void Main( string[] args )
   {
      // show that Count is 0 before creating Employees
      Console.WriteLine( "Employees before instantiation: {0}",
         Employee.Count );

      // create two Employees; Count should become 2
      Employee e1 = new Employee( "Susan", "Baker" );
      Employee e2 = new Employee( "Bob", "Blue" );

      // show that Count is 2 after creating two Employees
      Console.WriteLine( "\nEmployees after instantiation: {0}",
         Employee.Count );

      // get names of Employees
      Console.WriteLine( "\nEmployee 1: {0} {1}\nEmployee 2: {2} {3}\n",
         e1.FirstName, e1.LastName,
         e2.FirstName, e2.LastName );

      // in this example, there is only one reference to each Employee,
      // so the following statements cause the CLR to mark each
      // Employee object as being eligible for garbage collection
      e1 = null; // mark object referenced by e1 as no longer needed
      e2 = null; // mark object referenced by e2 as no longer needed
   } // end Main
} // end class EmployeeTest
```

Result



```
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
```

Object Initializers

Visual C# provides object initializers that **allow you to create an object and initialize its public properties** (and public instance variables, if any) in the same statement.

This can be **useful when a class does not provide an appropriate constructor to meet your needs**, but does provide properties that you can use to manipulate the class's data.

```
// create a Time2 object and initialize its properties
Time2 aTime = new Time2 { Hour = 14, Minute = 30, Second = 12 };
// create a Time2 object and initialize only its Minute property
Time2 anotherTime = new Time2 { Minute = 45 };
```