# CHAPTER 11

## MORE ADVANCED CLASSES

In this chapter, we shall explore the nature of C# classes more thoroughly. A key idea in object-oriented programming is that you can use the same classes in several applications. One such general-purpose class is a class named `Date` that will be introduced in this chapter, together with several programs that exploit the `Date` class. When people develop software systems that involve the use of classes, it is helpful if the class designs are described with graphical-textual drawings. This chapter introduces some basics of a systematic graphical-textual drawing method called the Unified Modeling Language (UML).

The last part of this chapter introduces properties and indexers which are class members that resemble methods in that they contain executable action statements. Properties are used to safely access data that is encapsulated inside objects. Indexers can make objects behave like arrays.

## 11.1 Class Date – an example of a larger class

One advantage of object-oriented programming is that the same classes can be used over and over again in many application programs. The C# programming environment, the .NET Framework, provides ready-to-use classes which you can exploit in your own programs. It is also important that you learn yourself to design general-purpose classes, and to use them in your programs. For this reason, we will study a general-purpose class named `Date` in this section, and you will be shown several programs that exploit the `Date` class. It is most important that you learn how class `Date` works. You do not necessarily have to understand every program line of the class, but it is important that you understand how an application program can create `Date` objects, and how the methods of the class work.

You have probably already guessed that class `Date` has something to do with presenting information about dates. To calculate time information in days, months, and years is not always such a simple thing to do. Some months have 31 days while others have only 30 days. Then there is the month of February which has only 28 days, except that once in every four years there is a leap year when February has 29 days. Then there is an exception that, although normally years equally divisible by four (e.g. 1992 and 1996) are leap years, years that are full centuries (e.g. 1800 and 1900) are not leap years. Then there is an exception to this exception that full centuries that are equally divisible by four (e.g. 1600, 2000, and 2400) are leap years. These are only some examples of the complexities of time calculation. Most of the complexities related to calculations of time as dates are incorporated in the methods of class `Date`. So this class should be useful when we need to handle date information in our programs.

The reasons why calculating with dates is complex are partly physical, partly historical. The historical reasons include the structure of our calendar and things such as how date information can be written down. In different countries people write dates in different ways. For example, in the United States, dates are written so that 10/18/2001 means the 18th day of October in year 2001. In Europe, it is common to write this date as 18.10.2001. Probably both of these styles to write dates are equally good, but the problem is that there is no single standard way. Class `Date` supports both of these date formats, and it can be made to support other formats if necessary.

The physical reasons why calculating with dates is difficult result from some astronomical facts. For example, a year is defined as the time during which the Earth goes around the Sun once. That is close to 365 and 1/4 days. A month approximates the time that the Moon uses to rotate the Earth once. Our programs for time calculation must be written so that they respect these and other astronomical facts.

Class `Date` solves many of the everyday problems related to calculations with date information. Three short programs demonstrate how class `Date` can be used in application programs. Program **Columbus.cs** shows how chronological distances between two `Date` objects can be calculated. Program **Birthdays.cs** shows how to easily find out what day of the week is a once-yearly date, such as a birthday. With program **Friday13.cs** you can help your superstitious friends. Program **Friday13.cs** prints a list of dates that are Fridays and the 13th day of a month. The class `Date` itself, which is exploited in all these example programs, is shown and explained as program **Date.cs**.

Class `Date` has three fields that are simple `int` variables to hold the day, month, and year of a `Date` object. The fourth field is `date_print_format` which gets either value 'A' or 'E', depending on whether a `Date` object ought to be printed in the American way MM/DD/YYYY or in the European way DD.MM.YYYY. Class `Date` has four constructor methods. `Date` objects can be created in different ways. For example, `Date` objects for the date 16th of August in year 2004 can be created in the following ways:

```
Date  first_american_date  =  new  Date( 16, 8, 2004, 'A' ) ;
Date  first_european_date  =  new  Date( 16, 8, 2004, 'E' ) ;
Date  second_american_date  =  new  Date( "08/16/2004" ) ;
Date  second_european_date  =  new  Date( "16.08.2004" ) ;
```

The last two ways to create **Date** objects are the easiest to use in practice. Both of these **Date** object creations invoke the same constructor method. The constructor examines the initialization date given as a string, and checks whether '/' or '.' is used to separate the numbers in the string. When '/' is used to separate numbers, the **Date** object becomes an American date, and it will later be printed in the American format. Objects initialized in the European way will later be printed in the European date format.

Class **Date** has almost twenty public methods. Four of these methods are short accessor methods which simply read the fields. For example, the method

```
public int day()  { return  this_day ; }
```

reads the protected field **this_day** and returns it to the caller. Methods like **day()** are commonly used in object-oriented programming, since, according to the principles of object-orientedness, data stored in objects should be protected from the outside world, and accessed only through methods. Methods that either read or write data fields can be labeled with the term "accessor method". Method **day()** is then a read accessor method as it allows a protected data field to be read. (Instead of accessor methods, it is possible to use so-called properties in C# classes. Properties will be discussed later in this chapter.)

The longer methods of class **Date** make various calculations related to time in days. The following list describes these methods:

- The boolean method **is_last_day_of_month()** returns **true** or **false**. This method is necessary because the months of a year have different lengths, and during leap years February has an extra day.
- The boolean method **this_is_a_leap_year()** contains the rules that specify whether a year is a leap year or not.
- Method **is_within_dates()** takes two **Date** objects as parameters. It returns **true** if the date for which the method was invoked is equal to or between the dates given as parameters.
- Method **index_for_day_of_week()** returns an integer in the range from 0 to 6. 0 means that the **Date** object is Monday, 6 meaning Sunday.
- Method **get_day_of_week()** calls method **index_for_day_of_week()** and returns a **string** object containing either "Monday", "Tuesday", ..., or "Sunday".
- The methods **increment()** and **decrement()** are used to rotate the dates stored in **Date** objects. These methods take care of leap years and varying lengths of months, so that **Date** objects are incremented and decremented correctly. These methods are called by several other methods of class **Date**.
- Method **get_distance_to()** calculates a chronological distance between two **Date** objects. **get_distance_to()** calculates the distance in whole years, months, and days.
- Method **get_week_number()** returns an integer that denotes the week of the year. Every **Date** object belongs to some week in the range from 1 to 53. Every year has at least 52 weeks. About every sixth year there is a year that has 53 weeks. The reason for this is that 52 weeks make only 364 days but years are either 365 or 366 days long. Week 53 is a kind of leap week that is used to consume the extra days that do not fit with the normal 52 weeks. Week numbers are commonly used in the calendars of many countries.
- Methods **is_equal_to()**, **is_not_equal_to()**, **is_earlier_than()**, and **is_later_than()** are methods of type **bool** that return **true** or **false** depending on what is the chronological relation between two **Date** objects.
- Method **ToString()** converts a **Date** object to a **string** object and returns it to the caller. When a class has a method with name **ToString()**, that method is invoked automatically in situations when objects of the class in question are joined to

`string` objects with the string concatenation operator +. Note that the name of this method is not `to_string()` but `ToString()`. The name must be `ToString()` in order to make the method automatically invoked by the compiler.

Class `Date` is written into its own source program file named **Date.cs**. When you use the class in a program that is in a different source program file, you have to compile the program so that you also mention the file name **Date.cs** in the compilation command. For example, programs **Columbus.cs**, **Birthdays.cs**, and **Friday13.cs** must be compiled with compilation commands like

```
csc Columbus.cs Date.cs

csc Birthdays.cs Date.cs

csc Friday13.cs Date.cs
```

because these programs use the `Date` class. In order to make the compilations succeed, all the **.cs** files that are mentioned in the compilation commands must be in the same directory (folder). If you compile your C# programs in the Microsoft Visual Studio .NET, you must include the **Date.cs** file into those projects which are created for programs **Columbus.cs**, **Birthdays.cs**, and **Friday13.cs**.

Class `Date` is a rather simple class that can be used when date information needs to be stored and handled in a program. Because the `Date` class is rather simple, it is a useful tool to study the nature of classes. However, the `Date` class is not a standard C# class, and therefore it cannot be recommended for wider use. C# has a standard class named `DateTime` that can be used to handle both date and time information. When you stop doing just programming exercises and start writing more serious programs, it is probably better that you learn to use the `DateTime` class. That class will be introduced later in this book.

### A first exercise with Date objects

Exercise 11-1. Write a program that calculates the chronological distance in years, months, and days from your birthday to any date that is given from the keyboard. You should, of course, use objects of class `Date` in this program. By studying program **Columbus.cs** you can find out how the chronological distance between two `Date` objects can be calculated. Program **Birthdays.cs** shows you how a date string can be converted into a `Date` object. You need the following kinds of statements in your program:

```
Date  my_birthday  =  new  Date( ...  // Your birthday here !

string  given_date_as_string  =  ...

Date  given_date  =  new  Date( ...
```

```
D:\csfiles3>Columbus

   Christopher Columbus discovered America on 10/12/1492
   That was a Wednesday

   Apollo 11 landed on the moon on 20.07.1969
   That was a Sunday

   America was discovered 476 years, 9 months, and 8 days
   before the first moon landing.
```

Those dates that are initialized in format MM/DD/YYYY are also printed this way.

**Columbus.cs - X. Outputting information related to dates.**

Class **Date** is declared in a separate C# source program file that is explained as a separate program description later in this chapter. You can use class **Date** in your program, when you include **Date.cs** in the compilation command.

The first date object that is created here is initialized with an American style date MM/DD/YYYY. The other **Date** object is initialized in the European way DD.MM.YYYY.

**Date** objects can be joined to strings with operator + because there is the method **ToString()** in class **Date**. The **ToString()** method is called automatically when operator + works as the string concatenation operator. Method **get_day_of_week()** returns either "Monday", "Tuesday", ..., or "Sunday", depending on what is the day of week of the **Date** object.

```
//  Columbus.cs  (c) Kari Laitinen

//  Compilation: csc Columbus.cs Date.cs

using System ;

class Columbus
{
   static void Main()
   {
      Date date_of_discovery_of_america = new Date( "10/12/1492" ) ;

      Date date_of_first_moon_landing  =  new Date( "20.07.1969" ) ;

      Console.Write(
          "\n   Christopher Columbus discovered America on "
        +  date_of_discovery_of_america  +  "\n   That was a "
        +  date_of_discovery_of_america.get_day_of_week() ) ;

      Console.Write(
          "\n\n   Apollo 11 landed on the moon on "
        +  date_of_first_moon_landing  +  "\n   That was a "
        +  date_of_first_moon_landing.get_day_of_week() ) ;

      int  years_between, months_between, days_between ;

      date_of_discovery_of_america.get_distance_to(
                                  date_of_first_moon_landing,
                                  out years_between,
                                  out months_between,
                                  out days_between  ) ;

      Console.Write( "\n\n   America was discovered "
                  +  years_between   +  " years, "
                  +  months_between  +  " months, and "
                  +  days_between    +  " days"
                  +  "\n   before the first moon landing.\n" ) ;
   }
}
```

Here, method **get_distance_to()** calculates the chronological distance from **date_of_discovery_of_america** to **date_of_first_moon_landing**. It writes the calculation result in variables **years_between**, **months_between**, and **days_between**.

**Columbus.cs - 1. Demonstrating the use of Date objects.**

```
//  Birthdays.cs  (c) 2003 Kari Laitinen

//  Compilation: csc Birthdays.cs Date.cs

using System ;

class Birthdays
{
   static void Main()
   {
      Console.Write( "\n Type in your date of birth as DD.MM.YYYY"
                  +  "\n or MM/DD/YYYY. Use four digits for the year"
                  +  "\n and two digits for the month and day:  " ) ;

      string  date_of_birth_as_string   =  Console.ReadLine() ;

      Date date_of_birth  =  new  Date( date_of_birth_as_string ) ;

      Console.Write(
         "\n   You were born on a "  +  date_of_birth.get_day_of_week()
       + "\n   Here are your days to celebrate. You are\n" ) ;

      int  years_to_celebrate  =  10 ;

      while ( years_to_celebrate  <  80 )
      {
         Date  date_to_celebrate  =  new  Date(

                     date_of_birth.day(),
                     date_of_birth.month(),
                     date_of_birth.year()  +  years_to_celebrate,
                     date_of_birth.get_date_print_format() ) ;

         Console.Write( "\n   "  +  years_to_celebrate
            +  " years old on "  +  date_to_celebrate
            +   " ("  +  date_to_celebrate.get_day_of_week()  +  ")" ) ;

         years_to_celebrate  =  years_to_celebrate  +  10 ;
      }
   }
}
```

This object creation results in a call to the fourth constructor that takes a string reference as a parameter. A date that is stored in a string is converted into a **`Date`** object.

As variable **`years_to_celebrate`** is incremented by 10 at the end of the loop, the program prints the dates for when the person is 10 years old, 20 years old, 30 years old, etc.

Here a new **`Date`** object is created each time the internal statements of the loop are executed. This statement invokes the **`Date`** constructor that takes four parameters. **`years_to_celebrate`** is always added to the birth year. **`day()`**, **`month()`**, **`year()`**, and **`get_date_print_format()`** are short methods that return the values of the corresponding fields. The values of the fields of an existing **`Date`** object are used to create a new **`Date`** object.

**Birthdays.cs - 1. A program that finds the dates for the most important birthday parties.**

```
D:\csfiles3>Birthdays

 Type in your date of birth as DD.MM.YYYY
 or MM/DD/YYYY. Use four digits for the year
 and two digits for the month and day:  14.07.1977

   You were born on a Thursday
   Here are your days to celebrate. You are

   10 years old on 14.07.1987 (Tuesday)
   20 years old on 14.07.1997 (Monday)
   30 years old on 14.07.2007 (Saturday)
   40 years old on 14.07.2017 (Friday)
   50 years old on 14.07.2027 (Wednesday)
   60 years old on 14.07.2037 (Tuesday)
   70 years old on 14.07.2047 (Sunday)
```

When you run this program on your own computer, it is important that you use leading zeroes when you give information of days and months. The program would not work if you wrote here 14.7.1977 or 7/14/1977. This advice also concerns program **Friday13.cs**.

**Birthdays.cs - X. The program is executed here with the input date July 14, 1977.**

### Some facts about our Gregorian Calendar

The calendar that is commonly used in most countries of the world is called the Gregorian Calendar because its development was initiated by Pope Gregory XIII. The Gregorian Calendar was taken into use in Roman Catholic countries in 1582, and within a couple of centuries the new calendar was in use in most European countries and the United States.

The calendar that was used before the Gregorian Calendar is called the Julian Calendar because it was developed and taken into use by following the orders of Julius Caesar. The problem with the Julian Calendar was that it had too many leap years because every fourth year was a leap year. This resulted in that, after a longer period of use, the Julian Calendar was behind the actual time. The problems of the Julian Calendar were corrected by the Gregorian Calendar that has more complex rules for calculating leap years (see method `this_is_a_leap_year()` in class `Date`). When the Gregorian Calendar was taken into use in 1582, 10 days were dropped from October. Thursday October 4 was followed by Friday October 15.

Because a new calendar has been taken into use since the days of Christopher Columbus, the information provided in program **Columbus.cs** is not entirely true. Columbus was using the Julian Calendar when he found America on October 12, 1492. That day is October 21, 1492, according to the Gregorian Calendar.

Although the Gregorian Calendar is the de facto official calendar of the present world, there are also other calendars in use. The Gregorian Calendar is not perfect either. One of its problems is that the week system of the calendar is not synchronized with the month system. Therefore the calendar is different every year. It is possible to specify calendars that are more stable than the Gregorian Calendar. Such calendars would make it easier to plan various activities in society. You can find information of proposed new calendars if you search the Internet with keywords like "calendar reform" and "world calendar".