M. E. Abramyan

USER INTERFACE DEVELOPMENT BASED ON WINDOWS FORMS CLASS LIBRARY

textbook

ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ЮФУ
РОСТОВ-НА-ДОНУ-ТАГАНРОГ

MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION
SOUTHERN FEDERAL UNIVERSITY

Mikhail E. Abramyan

USER INTERFACE DEVELOPMENT BASED ON WINDOWS FORMS CLASS LIBRARY

Textbook for students of computer science and programming

Rostov-on-Don – Taganrog
Southern Federal University Press
2021

УДК 004.438.NET
ББК 32.973.202
А16

Published by decision of the Educational-Methodical Commission of the I. I. Vorovich Institute of Mathematics, Mechanics, and Computer Science of the Southern Federal University (minutes No. 5 dated April 12, 2021)

Reviewers:

doctor of Technical Sciences, Professor of the Department of Computer Science of the Rostov State Transport University, Professor *M. A. Butakova*;

candidate of Physical and Mathematical Sciences, Head of the Department of Computer Science and Computational Experiment of the I. I. Vorovich Institute of Mathematics, Mechanics, and Computer Science of the Southern Federal University, Docent *S. S. Mikhalkovich*

Публикуется с разрешения издательства БХВ, являющегося владельцем прав на первое издание книги на русском языке: Абрамян М.Э., Visual C# на примерах, 2008.

Published with the permission of the publishing house BHV, which is the copyright holder of the first edition of the book in Russian: Абрамян М.Э., Visual C# на примерах, 2008.

Abramyan, M. E.

А16 User interface development based on Windows Forms class library : textbook / M. E. Abramyan ; Southern Federal University. – Rostov-on-Don ; Taganrog : Southern Federal University Press, 2021. – 278 p.

ISBN 978-5-9275-3830-0

The tutorial focuses on developing a graphical user interface based on the Microsoft Windows Forms class library, which is a part of the .NET Framework since version 1.0. The tutorial is presented in the form of detailed descriptions of 23 projects that demonstrate various aspects of user interface development for Windows applications. Projects can be implemented in the Microsoft Visual Studio 2015–2019 IDE. Description of projects is accompanied by numerous comments. Typical errors that arise during the development of Windows applications are considered, ways to fix them are indicated. The final section contains 65 study assignments designed to consolidate the learning material.

The textbook is intended for students specializing in science and engineering.

ISBN 978-5-9275-3830-0

УДК 004.438.NET
ББК 32.973.202

Contents

Preface

This textbook focuses on developing a graphical user interface based on the **Windows Forms** class library. This library appeared in .NET Framework 1.0 (the Microsoft Visual Studio .NET 2003 IDE), was significantly improved in .NET Framework 2.0 (the Microsoft Visual Studio 2005 IDE) and after that practically did not change; nevertheless, it was included in all subsequent versions of Microsoft Visual Studio. Despite the presence of the newer **Windows Presentation Foundation** class library, which is related to the development of the graphical interface and provides more features, the **Windows Forms** library retains its position, due to greater ease of use and convenient visual design tools. At the same time, the capabilities of this library are quite enough for the development of fully functional Windows desktop applications of medium complexity. In addition, the **Windows Forms** library contains a number of important concepts related to GUI design. Therefore, studying it in a course on the basics of user interface development seems to be quite justified.

The learning material of the book is presented in the form of descriptions of 23 examples, which are fully functional software projects. Each of the examples focuses on a specific topic indicated in its name, but much additional information is provided in the description of the project development process and accompanying comments. We focuse on best practices for developing event-driven applications and the efficient use of **Windows Forms** library components. The tools providing a convenient and reliable dialogue between the program and the user are discussed in detail. We also discuss common errors that occur when using the various **Windows Forms** library classes and show how to fix them.

Most of the examples are slightly modified versions of the examples given in [1]. The exception is the IMGVIEW example, which has been changed more significantly by eliminating several obsolete controls (DriveListBox, DirListBox, and FileListBox) and using the TreeView control instead, which allows visualizing hierarchical lists of data.

The book assumes an acquaintance with the basics of programming in the C# language at the level of knowledge of the system of basic types and control statements (see, for instance, [2]). The in-depth knowledge of the .NET object model is not required. Mostly C# 3.0 is used, which allows you to develop projects in the Visual Studio 2008 and higher. Among the innovations, only interpolated strings are used, which appeared in C# 6.0 (Visual Studio 2015); in earlier versions, you can use the string.Format function instead. The LINQ queries (implemented in C# 3.0) are used only in the projects LISTBOXES and IMGVIEW

(Chapters 19 and 21) and are commented in detail. Chapter 1 provides a detailed description of the Microsoft Visual Studio tools used to develop Windows applications.

The last chapter of the book contains 65 study assignments divided into 7 groups related to the following topics:

- developing console applications,
- interaction between windows of an application,
- synchronizing controls and sharing event handlers,
- implementing the drag-and-drop mode,
- creating timer-driven programs,
- use of standard dialog boxes and the Windows registry,
- developing MDI applications.

Most assignments contain references to sections of the book that describe the required controls and how to use them.

1. Developing projects in Microsoft Visual Studio environment

Currently, the most common version of Microsoft Visual Studio is version 2019, targeting the .NET 4.8 platform and the C# 8.0 language, although most of the features discussed are available for earlier versions. In particular, the **Windows Forms** library itself has remained unchanged since the release of .NET 3.5, C# 3.0, and Visual Studio 2008.

1.1. Creating, saving, and opening a project

When you start Visual Studio, the **Start Page** is automatically loaded into it, which allows you to quickly load one of the previously developed projects (the **Open recent** list), open any other existing project (the **Open a project or solution** item), and also create a new project (the **Create a new project** item). All these actions are also available from the menu of the Visual Studio environment:

- **File | New | Project…** or Ctrl+Shift+N – create a new project;
- **File | Open | Project/Solution…** or Ctrl+Shift+O – open an existing project;
- **File | Recent Projects and Solutions** – open one of the recently developed projects.

The Visual Studio environment is organized in such a way that you cannot create a "stand alone" project. Each project must be contained in a special entity called a *solution*, which can be described as a *group of related projects*. Only one solution can be loaded into the Visual Studio environment at a time; loading another solution leads to automatic closing of the previous solution.

When creating a new project, a dialog box appears on the screen, in which first of all you need to select the *language used* (in our case, **C#** or **Visual C#**), and then the *project template* and its name. As a template for all considered projects (except for the first two projects DISKINFO and EXCEP), you should choose **Windows Forms App** (or **Windows Forms Application**). As to the name, it is recommended to use the name of the example given in the title of the corresponding chapter, for example, EVENTS (see Chapter 4). Note that the project name can contain not only digits and Latin letters, but also other characters allowed in file names, including spaces, although this feature is not recommended for use.

In Visual Studio, when creating a new project, you must immediately specify the directory to save it (the **Location** text box). Project placement is also affected by information related to the solution in which the project will be placed,

such as the **Place solution and project in the same directory** checkbox. Note that previous versions of Visual Studio used a checkbox with the opposite meaning, **Create directory for solution**. If the solution includes a single project, then you should check the **Place solution and project in the same directory** checkbox (or uncheck the **Create directory for solution** checkbox); this will cause the created solution to have the same name as the created project, and all the files associated with the project and its solution will be placed in the same directory specified in the **Location** text box.

If the **Place solution and project in the same directory** checkbox is unchecked (or the **Create directory for solution** checkbox is checked), you can specify a solution name that may be different from the project name. In this situation, a more complicated directory hierarchy is created: a solution directory will be created in the directory with the name specified in the **Location** text box and the project directory will be created in the solution directory.

Note one more feature of the Visual Studio environment (which we, however, will not use): the created project can be *added* to the current solution.

To save all changes made to the current project (or rather, to the current solution), Visual Studio provides the **File | Save All** menu command, as well as the Ctrl+Shift+S shortcut key.

When you open an existing project, the **Open** dialog box appears on the screen, in which you can select either a file with the .**sln** extension (containing information about the solution with the specified name) or a file with the .**csproj** extension (containing information about the project with the specified name). However, even if you select some project, the entire solution project containing it will be loaded.

Examples TEXTEDIT2 – TEXTEDIT6 suggest not creating a new project, but modifying an existing project. Before starting to develop such examples, you should copy all the solution files from the example with the previous number to a new directory, then load this resulting copy into Visual Studio and start modifying it.

Let us describe the steps for modifying projects using the example of converting the TEXTEDIT1 project into the TEXTEDIT2 project.

First, you should change the name of the project and the name of the associated solution. The simplest way to perform this is using the **Solution Explorer** window, usually located on the right side of the screen (Fig. 1.1 on the left shows this window for the TEXTEDIT1 project): just select the line corresponding to the project in the **Solution Explorer** window (in Fig. 1.1, this line is highlighted) or a solution (this line begins with the word **Solution**), press the F2 key, enter a new name and press Enter. The same action should be performed for the name of the solution. The **Solution Explorer** window after changing the name **TEXTEDIT1** of the project and its solution to **TEXTEDIT2** is shown in Fig. 1.1 on the right.

You should also change the *assembly name*, that is, the name of the resulting exe-file. To do this, run the **Project | <Project name> Properties** menu command (in our case, **Project | TEXTEDIT2 Properties**), go to the corresponding editor tab with the project name, select the **Application** section of this tab, and specify a new name in the **Assembly name** text box (Fig. 1.2). You can also change the default namespace name for this project (the **Default namespace** text box), but this is not necessary.

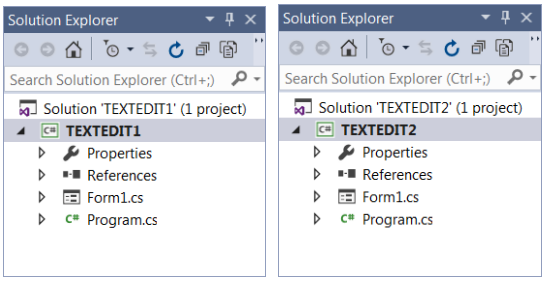


Fig. 1.1. **Solution Explorer** window before changing the project name (left) and after changing it (right)

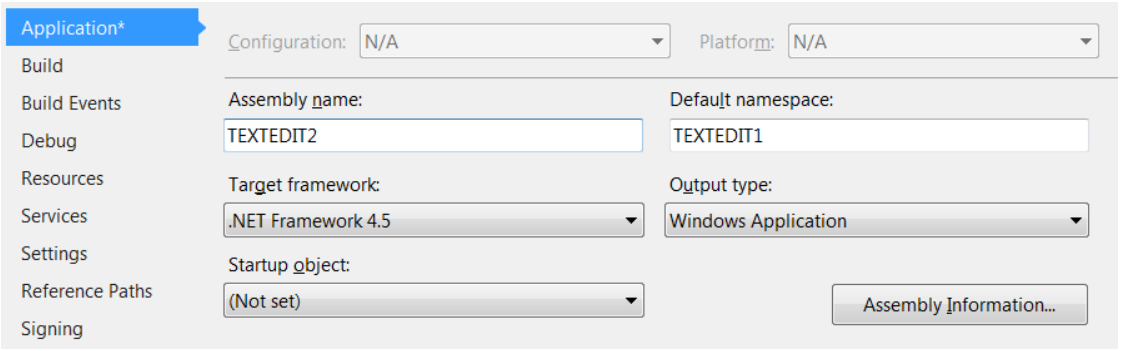


Fig. 1.2. Top part of the project properties window

1.2. Adding a new form to the project and placing a new control on the form

When you create a new **Windows Forms Application** project, its *main form* (a class named Form1) is automatically created and immediately loaded into Visual Studio. A set of files is associated with Form1, the main of which are **Form1.cs** and **Form1.Designer.cs**. These files contain a description of this form in C#, and the second of them (**Form1.Designer.cs**) contains the part of the description that is generated automatically in response to various developer actions related to visual design (the developer's own code is usually placed in the **Form1.cs** file).

If you need to use additional forms in your project (see, for example, Chapter 5), then the easiest way to add them to the project is by using the **Project | Add Form (Windows Forms)…** menu command (in previous versions, the command was named **Project | Add Windows Form…**). When this command is executed, a dialog box appears in which you must specify the name of the added form (which will also be the initial part of the names of the files contain-

ing its description). The examples described always use the default form names (Form1, Form2, etc.).

Any created form is displayed in the Visual Studio editor in *design mode* (an image of the form and its contents appears on the screen); the corresponding editor tab name ends with the text **[Design]**. For example, for the main form Form1, the tab name contains the text **Form1.cs [Design]**. To go to the contents of the corresponding cs-file, just press the F7 key; a new tab with the text of this file will appear in the editor (the name of this tab will contain the file name, for example, **Form1.cs**). To switch back from the program code to the form image, you can use the Shift+F7 key combination. You can also switch between the form image and the program code using the context menu: when you right-click on the image of the form, the first item of the pop-up menu is named **View Code** and allows you to go to the program code of the cs-file, and when the context menu for the program code of the cs-file is called, the **View Designer** menu item is available which allows you to return to design mode.

Another convenient way to switch between different windows of the Visual Studio environment is to use the Ctrl+Tab key combination: after pressing it, an auxiliary window appears on the screen with a list of all loaded files and forms (**Active Files**), as well as all open auxiliary tool windows (**Active Tool Windows**). Fig. 1.3 shows an example of this window. After this window appears, you should, without releasing the Ctrl key, select the name of the file or tool window that you want to activate (you can use the arrow keys and the Tab key to select), and then release the Ctrl key.

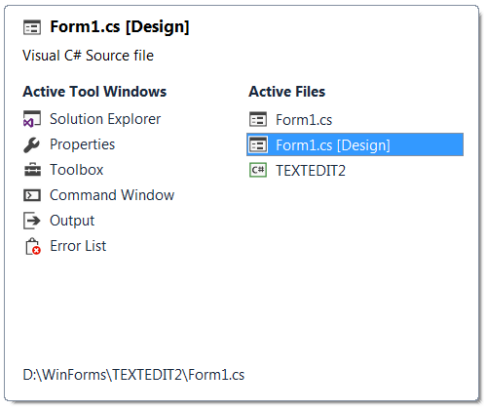


Fig. 1.3. Auxiliary window for selecting one of the loaded files or tool windows of the Visual Studio environment

Any tab in the editor can be closed; to do this, just make it active and press the Ctrl+F4 key combination or call the context menu of a tab by right-clicking on its tab name and select **Close**.

If the required form is not loaded in the editor after loading an existing project, it is enough to double-click the name of this form in the **Solution Explorer** window (see Fig. 1.1). Alternatively, you can right-click the name of this form and select **Open** from the context menu that appears. This menu also contains

the **View Code** item, which allows you to load the text of the cs-file into the editor, and the **View Designer**, which allows you to load the form image into the editor in design mode.

To place a new control on the form, use the **Toolbox** window (this window can be displayed on the screen either by the **View | Toolbox** menu command or by the Ctrl+Alt+X key combination). This window is usually displayed on the left side of the screen; Fig. 1.4 contains a view of the upper part of the **Toolbox** window when the **All Windows Forms** group is selected in it. Note that controls appear in the **Toolbox** window only when the editor is in design mode. In this window, it is necessary to select the group containing the required control (by clicking on the name of this group), then click on the name of the required control to select it (in Fig. 1.4, the **Button** control is selected), and finally click in the position of the form where the selected control is supposed to be placed.

In order to select a control in the **Toolbox** window, you do not need to know which group it is contained in, since this window contains the **All Windows Forms** group, which lists all available controls in alphabetical order (see Fig. 1.4).

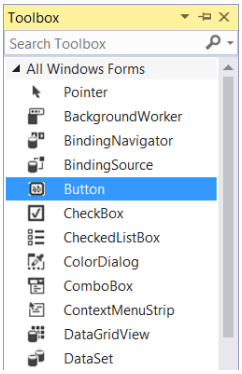


Fig. 1.4. Top view of the Toolbox window

If the **Toolbox** window covers the part of the form where you want to place the control, you can click on the image in the title bar of the **Toolbox** window; this will cause the **Toolbox** window to be positioned to the left of the form window, and the image will be changed to . Note that, in this mode, the **Toolbox** window will not automatically hide after placing the selected control on the form (to hide the **Toolbox** window, just click on the image).

To quickly place *several* controls of the same type on a form, you should, after selecting the required control in the **Toolbox** window, click the form several times while holding down the Ctrl key. If you need to deselect a control in the **Toolbox** window without placing it on the form, then it is enough to select the **Pointer** item in this window (it is located first in any group of controls – see Fig. 1.4).

If you want to place a control not on a form, but in another control (for example, in a **Panel** control), then, when placing the control, you must click in the

area of the target control. Only special controls called *container controls* can be used as target controls (the form itself is a container control too). A container control (form, panel, etc.) that contains other controls is called the *parent* of the *child* controls it contains.

When describing the stages of project development, we will always indicate in which container control each control should be placed.

You can display a *parent-child hierarchy* of all controls located in a form. The corresponding window is displayed on the screen by the **View | Other Windows | Document Outline** menu command. Using the drag-and-drop of the controls in this window, you can also adjust their hierarchy. See Fig. 21.7 in Section 21.2 for an example of the **Document Outline** window.

After adding a control to a form, it automatically gets a name, which is stored in its Name property. By default, the name of any control starts with a lowercase letter and consists of the control type name and an order number. Thus, the first button placed on the form (a control of type Button) will be named button1, the second button will be named button2, and so on. Forms have names that begin with an uppercase letter (Form1, Form2, etc.); this is due to the fact that these names are the names of new *classes* that are descendants of the base Form class. The default name can always be replaced with another name that allows, for example, to clarify the purpose of a particular control (for example, the button containing the text **OK** can be named as btnOK). However, the described projects almost always use the control names offered by the Visual Studio environment by default; this allows you to reduce the number of steps for setting control properties and simplifies reading program code. "Meaningful" names are used only for menu items, shortcut buttons, and sections of the status bar (see Chapters 12, 15, 16). It should be noted, however, that when developing large projects it is advisable to use meaningful names for all controls.

When describing actions for adding a control to a form, it is almost never specified how to position a control on its parent control, since this can be easily determined from the given figure. Let us list the ways to position a control:

- dragging with the mouse over the form (while auxiliary alignment lines appear on the form allowing the control to snap to the boundaries of the form or place it at the level of the boundaries of existing controls);
- using the alignment panel **Layout** (see Section 18.1, Comment 1);
- move by one pixel using the arrow keys;
- explicit indication of the value of the Location property in the **Properties** window (working with the **Properties** window is discussed in detail in Section 1.3).

The sizes of the visual controls are also usually not specified. To adjust the control size, you can drag with the mouse on one of the handles surrounding the selected control, or use the arrow keys while holding down the Shift key. You

can also explicitly set the Size property in the **Properties** window or use the **Layout** panel.

For more information related to customizing the application menu, its toolbar, and status bar, see Chapters 12, 15, 16.

1.3. Setting properties of forms and controls

To configure the properties of forms and controls, use the **Properties** window, which can be quickly accessed using the Alt+Enter key combination (you can return back to the form image using the previously described Ctrl+Tab combination). The **Properties** window is usually located in the lower right corner of the screen and can be in two modes: **Properties** and **Events** (see Fig. 1.5). In this section, we will consider the **Properties** mode designed to display the properties of a selected control or a group of controls (to switch to this mode, click the third button on the toolbar of the **Properties** window).

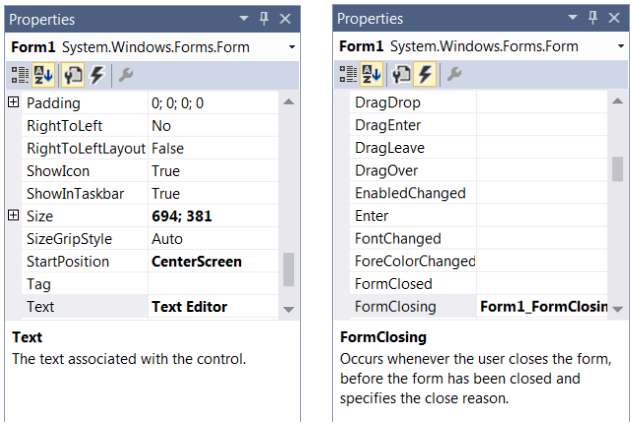


Fig. 1.5. **Properties** window in **Properties** (left) and **Events** (right) modes

If a group of controls is selected, then the properties window displays only those properties that are available for all selected controls.

Selecting a group of controls allows you to quickly set them to the same size, headers or other common properties, as well as move them, while maintaining their relative position. Let us list the ways to select a group of controls:

- clicking on the controls while holding down the Shift or Ctrl key;
- coverage of controls by a dotted frame that appears on the form when moving the mouse with the left button pressed (to select a control, it is enough to grab a *part* of it with the frame).

If a group of controls is selected, then one control of this group is *current* (its markers are white). Fig. 1.6 shows a fragment of a form with four selected controls; the current control is button2. The result of some actions (for example, alignment of controls on a form – see Section 18.1) depends on which control of the selected group is the current one. To make another control from the selected group the current one, just click on it with the mouse. To deselect a group of controls, select a control that is not included in this group.

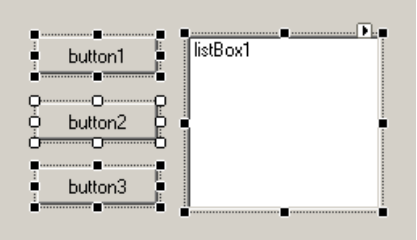
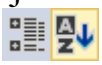


Fig. 1.6. Fragment of a form with a group of selected controls

It is useful to know that if a control is selected, then, to select its parent control, it is enough to press the Esc key (in this way, you can quickly select any container control, even if it is entirely covered by child controls). There is an even easier way to select a form: just click on its *title bar*.

Using the first two buttons  of the **Properties** window (see Fig. 1.5), you can configure how the properties are displayed: by *category* (the first button) or in *alphabetical order of names* (the second button). Each category has a name (for example, **Appearance**) and contains "related" properties. By grouping properties alphabetically, it is easier to navigate to the desired property; at the same time, at the initial acquaintance with the properties available in the control, it is more convenient to use grouping by categories.

In the example projects, setting the properties of forms and controls is described in the listings entitled **Properties**. An example of such a listing is the following listing taken from Section 5.1:

Properties

```
Form1: Text = Main Window, MaximizeBox = False,
  FormBorderStyle = FixedSingle
button1: Text = Open subordinate Window
button2: Text = Open dialog window
```

In the property setting listings, we first specify the name of the control whose properties we want to change and then, after the colon, a list of its custom properties in the format

```
property name = new property value
```

(the property value is in **bold**). Properties are listed separated by commas.

Controls can have *composite properties*, that is, properties that have their own properties (for example, Font). If we need to set one or more properties of such a composite property, the dot separator is used in the property setting listing, for example (this is a copy of listing from Section 23.2):

Properties

```
Form2: Text = empty string, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%,
  ShowInTaskbar = False, StartPosition = CenterScreen,
  UseWaitCursor = True
label1: Text = Trigonometric Functions,
```

```
AutoSize = False, Dock = Fill, TextAlign = MiddleCenter,
Font.Name = Times New Roman, Font.Size = 32, Font.Bold = True
```

Composite properties are marked in the **Properties** window with a "+" mark to the left of the name (in Fig. 1.5, the Padding and Size properties contain the "+" marks). Clicking on the "+" mark displays all the properties of the selected composite property, and the "+" mark changes to "–" mark. Clicking on the "–" mark collapses the list of properties of the composite property.

The previous property setting listing demonstrates another notation used: if a property needs to be cleared, the italic text "*empty string*" is used as its value.

The Boolean property values use the constants **True** and **False** that start with a capital letter (as opposed to the C# keywords true and false), because this is how Boolean constants are specified in the **Properties** window.

The **Properties** window uses sub-panels to set some anchoring or alignment properties (such as Dock or TextAlign). To display such panels, use the button  that appears when the corresponding property is selected. In these panels, you need to select one or several elements located in the required position (left, right, center, etc.); the result of the selection will be displayed in the **Properties** window as plain text (for example, **Bottom** or **MiddleCenter**). Although these actions are intuitive, they are usually provided with additional explanations in the description of example projects.

In some cases, it is convenient to use a special *dialog box* to set a property. If a property has such a dialog box, then, when the property is selected in the **Properties** window, a button  is displayed to the right of it, which allows you to call a dialog box (an example of such a property is Font).

1.4. Defining event handlers

All controls have not only a set of properties, but also a set of *events* that can be assiciated with methods called *event handlers*. The list of handlers for the selected control or group of controls is displayed in the **Properties** window in the **Events** mode (see Fig. 1.5); to switch to this mode, press the fourth button  on the toolbar of the **Properties** window. The list of events, like the list of properties, can be ordered in two ways: by *category* and by *name* (in alphabetical order).

Double-clicking on the empty input text box next to the name of the required event automatically creates a template for the event handler. Any event handler for any control is a *method of the form* on which the control is located. All examples use the default names of the handler methods provided by Visual Studio to make it easy to determine which control and event type the handler is associated with.

The handler texts are shown in listings whose headings end with the word "handler", for example (this is a copy of the event handler from Section 4.1):

Form1.MouseDown handler

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  button1.Location = new Point (e.X - button1.Width / 2,
    e.Y - button1.Height / 2);
}
```

The text that you should add to the automatically generated handler method template is in **bold**. Sometimes program fragments are provided with comments, although more often explanations are given in the main text of the example or in a special **Comment** section.

If you need to use an *existing* method as a handler, then just select the name of this method in the drop-down list next to the event name in the **Properties** window (the expanding button appears near the selected event).

Since, when developing programs in Visual C#, you need to have a clear understanding of how event handlers are created and used, a special project EVENTS (Chapter 4) is devoted to this topic. Connecting a handler to multiple events is discussed in detail in Chapter 6.

1.5. Making changes to the program text

When developing projects, changes are usually made to the text of existing handlers. If the changes are significant and cover the entire text of the handler, then its new full text is shown, in which the added lines or line fragments are highlighted in **bold** and removed lines are ~~strikethrough~~, for example (this handler is a copy of the handler from Section 22.7):

```
private void Form2_FormClosed(object sender,
  FormClosedEventArgs e)
{
  Form1 f = MdiParent as Form1;
  if (MdiParent.MdiChildren.Length == 1)
    (MdiParent as Form1).window1.Visible = false;
    f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
      f.zoom2.Enabled = false;
}
```

If the changes are insignificant and the handler is large enough, then it is simply indicated which statements need to be added or replaced, for example:

Add the following statement to the zoom1_CheckedChanged method:

```
(MdiParent as Form1).zoom2.Checked = zoom1.Checked;
```

Change the last statement in the Form2_FormClosed method as follows:

```
f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
  f.zoom2.Enabled = f.zoom2.Checked = false;
```

If the place of addition is not specified, then statements should be appended to the *end* of the method.

You can go to an existing handler to modify it either by moving through the text of the file or using the **Properties** window by double-clicking on the text box with the name of the required handler.

Changes to the program code that are not associated with a specific handler are described in a similar way, for example:

Add the field to the description of the Form1 class

```
private Form2 form2 = new Form2();
```

Add the statement to the constructor of the Form1 class:

```
AddOwnedForm(form2);
```

When typing and editing program code, you should use the additional features of the Visual Studio editor. Let us describe some of them.

1. If you type the name of an object (for example, button1) and a period "." after it, then a list of all methods and properties that this object has will appear on the screen, and to quickly select the desired method, you just need to type its first few characters. To insert the name of the selected method or property into the program code, press the Enter, Tab or Spacebar keys. The list of methods and properties can also be called explicitly by pressing the Ctrl+Spacebar key combination.

2. After typing the method name and parenthesis "(", a prompt appears on the screen with a short description of this method and a list of all its parameters. If the method is overloaded, that is, it can be called with a different set of parameters, then you can browse all of its overloaded versions by pressing the keys Up (↑) and Down (↓). You can also press Ctrl+Shift+Spacebar to show this hint.

3. It is convenient to use *bookmarks* to quickly jump to the desired piece of code. To *set/clear* a bookmark on the current line of the program code (that is, the line containing the keyboard cursor named *caret*), press Ctrl+K and then again the same Ctrl + K key combination. To go to the *next* or *previous* bookmark, press Ctrl+K and then Ctrl+N or Ctrl+P, respectively. You can also use the **Edit | Bookmarks** menu group and speed buttons  in the **Text Editor** panel.

4. If you need to *comment out* any piece of code, then just select it and press Ctrl+K and then Ctrl+C. In order to *uncomment* a commented out a piece of code, you need to select it and press Ctrl+K and then Ctrl+U. If you need to comment out or uncomment one line, then it is enough to place the caret on it (instead of selecting it) and then press the indicated key combinations. Instead of shortcut keys, you can use the speed buttons  in the **Text Editor** panel.

5. The editor of the Visual Studio environment has rich *search and replace tools*. These tools are described in detail in Comment 2, Section 9.5.

1.6. Application launch

Each stage of project development is described in a separate section of the chapter connected with that project. Each section begins with a description of

the steps involved in changing the project. This is followed by a paragraph beginning with the **Result** word. This paragraph describes how the new version of the program will work. The presence of the **Result** paragraph is a sign that the modified project can be compiled and run (to do this, just press the F5 key or the button ).

If the program code contains syntax errors, then messages about these errors appear in the **Error List** window, which becomes active. In order to go to the program line in which the first syntax error was found, just press the Down (↓) key (as a result, the message about the first error will be highlighted) and then press Enter. Alternatively, you can double-click on the line with the error message.

When compiling a program in the Visual Studio environment, two environment settings play an important role. These settings are located in the **Projects and Solutions** group of the **Options** window (this window is invoked by the **Tools | Options...** menu command). The first setting is in the **General** section; this is the **Always show Error List if build finishes with errors** checkbox that must be selected. The second setting is in the **Build and Run** section; this is the **On Run, when build or deployment errors occur** drop-down list in which the **Do not launch** option should be selected.

If the **Result** paragraph is followed by a paragraph marked with the **Error** word, it means that the program, despite successful compilation, will not work quite correctly, and additional corrections must be made to it (this allows you to draw attention to typical errors that can occur in similar situations). If the **Result** paragraph is followed by a paragraph marked with the **Disadvantage** word, then this means that the program is working correctly and doing what is required, but it has interface defects, that is, it is inconvenient to use. As a rule, immediately after the description of the error or disadvantage , the way of correcting them is indicated, although sometimes the correction is postponed until the next section.

2. Console application: DISKINFO project

The DISKINFO project introduces techniques for developing *console applications*. We describe the structure of a console application and the namespaces connected to it. Formatted output is discussed, in particular the use of escape sequences. The DriveInfo, StringBuilder, and Environment classes are described. Also we examine issues related to the use of command line arguments.

2.1. Creating a console application

When creating a new *console application* project, you should follow almost the same steps as when creating a *Windows application* project (see Section 1.1). The only difference is specifying a different project template in the **New Project** window: for a console application, select the **Console App** (or **Console Application**) template.

The created project contains the **Program.cs** file, which will be loaded into the editor:

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DISKINFO
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

The first five directives in the **Program.cs** file contain the names of the *namespaces* with most commonly used standard classes (see Comment 1). Due to these directives, class names can be used in the program code without specifying namespace they belong to (for example, instead of System.Console, you can simply write Console).

The Main method is the starting point for program execution. Its args parameter allows you to get information about the *command line arguments* speci-

fied when launching this program (the use of the args parameter is demonstrated in Section 2.3).

Add new statements to the Main method:

```
Console.WriteLine("DISKINFO program\n");
Console.WriteLine("\nPress <Enter> to end the program...");
Console.ReadLine();
```

Result. When the program starts, a special *console window* appears on the screen, which is used for data input-output in text mode. The console window contains the following text:

```
DISKINFO program

Press <Enter> to end the program...
```

After the program terminates, the console window is closed immediately. To give opportunity to acquaint with the contents of the window, a call to the ReadLine method of the Concole class has been added to the program (see the last statement). This method is intended for input a string, and the sign of completion of the input is the pressing of the Enter key. Therefore, until you press Enter, the console window will remain on the screen. The string returned by the ReadLine method is not used in the program, so there is no need to store it in any variable.

Comments

1. Let us give a brief description of the namespaces automatically connected to our application (see the first five directives). The System namespace is the primary namespace for the .NET Framework standard library; it contains class definitions that are needed in almost any program. In particular, it defines the Concole class, which provides input-output for the console window. The System.Collections.Generic namespace contains *generic collection classes*. Generic classes were introduced in .NET 2.0; the way of implementing collections based on generics allows you to specify when declaring a collection what type of data it will contain (an example of using a generic collection class is given in Section 11.1). The System.Linq namespace allows you to use *LINQ queries* in your program to handle various *data sequences* including arrays and collections (an example of using LINQ queries is given in Section 19.3). The System.Text namespace contains various classes and enumerations for handling text data; in Section 2.2, we will use the StringBuilder class defined in this namespace. The System.Threading.Tasks namespace contains classes for including *parallel* and *asynchronous* code into a program (these classes will not be used in the book).

2. The \n combination used in string constants is one of the *escape sequences* that can be specified in string expressions. This combination denotes character with code 10 (a *new line*). Some other escape sequences are:

- \\ (character “\”),

- \" (double quote),
- \' (single quote),
- \0 (character with code 0),
- \r (*carriage return* – character with code 13),
- \b (character with code 8 generated by the Backspace key),
- \t (*tabulation* – character with code 9),
- \u*N* (*Unicode character* with hexadecimal code *N*).

In the escape sequence \u*N*, code *N* consists of 4 hexadecimal digits and can be 0000 to FFFF; for example, the character \u0041 denotes the Latin letter A (code 65).

3. If you run the console application not in **Debug** mode, but in **Release** mode (by pressing Ctrl+F5 instead of F5), then, when it finishes, the text appears in the console window "Press any key to continue..." and the window will be closed only after pressing an arbitrary key. Thus, in this mode there is no need for the Console.ReadLine() statement (moreover, if this statement is present, to close the application in **Release** mode, you will first have to press Enter, and then some arbitrary key). However, when you run the resulting exe-file directly from Windows (not from Visual Studio), the application runs in the same way as in **Debug** mode, that is, the "Press any key to continue..." message does not appear.

4. Starting from version C# 6.0 (.NET Framework 4.6, Visual Studio 2015), you can omit the name of the Console class when calling its methods if you first specify a special version of the using directive:

```
using static System.Console;
```

After the using static text, you must specify the *fully qualified name* of the class (that is, the name that includes the namespace), which you can later omit when calling its methods (for example, instead of the Console.WriteLine() or Console.ReadLine() statement, you can just specify WriteLine() or ReadLine()).

2.2. Receiving the information about current disk

At the top of the **Program.cs** file, add the statement

```
using System.IO;
```

Add a new DInfo method to the Program class:

```
static void DInfo(string path)
{
  string none = "---",
    d = path[0].ToString().ToUpper();
  DriveInfo di = new DriveInfo(d);
  StringBuilder s = new StringBuilder(40);
  s.AppendFormat(" {0,-4}", d);
  if (di.DriveType != DriveType.NoRootDirectory)
```

```
  {
    s.AppendFormat(" {0,-9}", di.DriveType);
    if (di.IsReady)
      s.AppendFormat("{0,14:N0} {1,14:N0}", di.TotalSize / 1024,
        di.TotalFreeSpace / 1024);
    else
      s.AppendFormat("{0,14} {0,14}", none);
  }
  else
    s.AppendFormat(" {0,-9}{0,14} {0,14}", none);
  Console.WriteLine(s);
}
```

Change the Main method as follows:

```
static void Main(string[] args)
{
  Console.WriteLine("DISKINFO program\n");
  Console.WriteLine(" Disk Type         Size (K)      Free (K)");
  Console.WriteLine(new String('=', 40));
  DInfo(Environment.CurrentDirectory);
  Console.WriteLine("\nPress <Enter> to end the program...");
  Console.ReadLine();
}
```

Result. When the program is launched, information about the current disk is displayed in the console window, for example:

```
DISKINFO program

 Disk Type           Size (K)       Free (K)
========================================
 D    Fixed       720 759 804     14 519 580

Press <Enter> to end the program...
```

Comments

1. The DInfo method takes a path string as a parameter, from which only the first character is used (it is assumed that this character is a *drive letter*). Since path[0] is a char expression, to be able to use this expression as a string parameter to the constructor of the DriveInfo class, we must explicitly convert it to string using the ToString method. The resulting single-character string is then converted to uppercase using the ToUpper method of the string class.

2. The DriveInfo class is used to retrieve disk information. This class is introduced in .NET 2.0 Framework and defined in the System.IO namespace. To create an object of the DriveInfo class, just call its constructor specifying the let-

ter of the required drive. The DriveType property (of DriveType enumeration type) allows you to determine the type of disk media, even if the disk is unavailable (among the possible values of the DriveType enumeration, we indicate Removable, Fixed, Network, and CDRom). For an available disk, we can determine its size in bytes (the TotalSize property of long type), the size of free space in bytes (the TotalFreeSize property of long type), and its label (the VolumeLabel property of string type; this property is available for both reading and writing).

Note that the long type used for the TotalSize and TotalFreeSize properties allows numbers to be stored in the range from –9 223 372 036 854 775 808 to 9 223 372 036 854 775 807 (thus, it is possible to determine the size of a disk containing more than *9 million terabytes*).

3. When forming a string with information about the disk, the s object of the StringBuilder class was used. This class is defined in the System.Text namespace and allows you to more efficiently (compared to the string class) perform string operations. In the constructor of the StringBuilder class, we specified the *capacity* of the generated string (that is, its maximum possible size of 40 characters), but this does not mean that the s object cannot contain larger strings: if the size of the actual string stored in an object of StringBuilder type exceeds its capacity, the capacity is automatically doubled. Explicit indication of capacity in the constructor avoids unnecessary memory allocation and deallocation.

An important feature of objects of StringBuilder type, in comparison with objects of string type, is the availability of their symbols not only for reading, but also for writing. In addition, any actions to change a string of StringBuilder type are performed on the same string and do not lead to the creation of a new (changed) string, as it happens in the string class methods that modify string data. The AppendFormat method used in the program adds new data to the previous content of the string and formats the data.

4. When formatting the data, special formatting settings were used, which generally have the form {ind,width:spec}, where ind defines the *index* of the formatted element in the subsequent list of parameters (indexing is carried out from 0), width defines the *minimum* output field width for the formatted element and the way it is aligned within this field (if width is positive, then alignment is performed on the right, if negative, then on the left). The spec attribute specifies the *format specifier* for this element. The N specifier used in the program allows to display a number with spaces (thousands separators); the value 0 specified after it means that the number of fractional characters is zero (by default, N format displays *two* fractional characters). Let us list some other format specifiers:

- C (currency format),
- D (decimal integer format),
- X (hexadecimal integer format),

- E (exponential number format),
- F (fixed-point number format),
- P (percentage format).

Format specifiers can be used in many methods related to string formatting, such as the Format method of the string class and the Write and WriteLine methods of the Console class. Of the attributes included in format settings, only the ind attribute is required; it is permissible to specify several *identical* values for the ind attribute if you want to display the same data item several times (see the last two calls to the AppendFormat method in the DInfo method definition. If the width attribute is absent, the minimum width of the output field is used to display the formatted item. If the spec attribute is missing, then the default formatting option is selected (this option corresponds to G format specifier). If there is no width, no leading comma (,) is specified; if there is no spec, no leading colon (:) is specified.

5. Instead of explicitly specifying a string containing 40 characters "=" (equal sign), we used a version of the string constructor with two parameters (c, n). This constructor allows creating a string of the specified length n with identical characters c.

6. To determine the *current disk*, the CurrentDirectory property of the Environment class was used, which allows you to get the current directory for this application and is available for both reading and writing. Other useful properties of this class are CommandLine, which returns a string with the full name of the running exe-file followed by command line arguments, SystemDirectory, which returns the full name of the Windows system directory, a set of properties that allow you to get information about the computer, user, and operating system: MachineName, UserName, OSVersion, Version (the last property returns the full .NET Framework version in use).

There is also an array property GetCommandLineArgs of string[] type, the first element of which (with index 0) contains the full name of the exe-file and each next element contains the next command line argument (thus, the GetCommandLineArgs property is a CommandLine string parsed into separate words).

In addition, the Environment class allows you to get and change the values of various *environment variables* defined in the operating system. To do this, it has the following methods: GetEnvironmentVariables (returns an array of all environment variables as a name–value string pair), GetEnvironmentVariable(name) (returns a string value for an environment variable named name or null if the specified environment variable does not exist), SetEnvironmentVariable(name, val) (sets an environment variable named name to the new string value val). If the variable with the name specified in the SetEnvironmentVariable method does not exist, then it is created; if val is an empty string or null, then the existing variable with the given name is destroyed.

7. Starting with C# 6.0, you can format expressions included in strings without using the special Format methods described in Comment 4: you just need to include the required expression in the string itself, enclose it in curly braces, and provide optional formatting attributes using the rules described in Comment 4. Strings formatted in this way are called *interpolated strings*; before their opening double quote ", you must specify the $ symbol. As an example, we will give a part of the DInfo function (a block of statements after the header of the external if statement), which uses interpolated strings:

```
s.Append($" {di.DriveType,-9}");
if (di.IsReady)
  s.Append($"{di.TotalSize /
    1024,14:N0} {di.TotalFreeSpace / 1024,14:N0}");
else
  s.Append($"{none,14} { none,14}");
```

2.3. Using command line arguments

Modify the Main method in the **Program.cs** file as follows:

```
static void Main(string[] args)
{
  Console.WriteLine("DISKINFO program\n");
  Console.WriteLine(" Disk Type         Size (K)      Free (K)");
  Console.WriteLine(new String('=', 40));
  if (args.Length == 0)
    DInfo(Environment.CurrentDirectory);
  else
    foreach (string d in args)
      DInfo(d);
  Console.WriteLine("\nPress <Enter> to end the program...");
  Console.ReadLine();
}
```

Result. If one or more drive names are specified as command line arguments, information about these drives is displayed (command line arguments must be separated from each other by spaces). If arguments are not specified, then information about the *current disk* is displayed. To set command line arguments in Visual Studio, perform the **Project | DISKINFO Properties…** menu command, select the **Debug** section in the project properties tab loaded into the editor and enter the command line arguments in the **Command line arguments** text box. So, if you specify **d e f z** as the command line arguments, then, as a result of the program execution, a text similar to the one below will be displayed:

```
DISKINFO program

 Disk Type           Size (K)       Free (K)
=======================================
 D    Fixed       720 759 804     14 519 580
 E    Fixed     2 930 265 084     16 317 180
 F    CDRom               ---            ---
 Z    ---                 ---            ---

Press <Enter> to end the program...
```

In the Windows environment, the program with parameters can be launched, for example, from the **Start** menu using the **Run…** command:

```
D:\Apps\DISKINFO\bin\Debug\DISKINFO.exe d e f z
```

If the program is launched using a *shortcut* (that is, a special file with the .**lnk** extension), then its command line arguments can be set in the shortcut properties window, which is invoked by the **Properties** command of the context menu of the shortcut. In the properties window, go to the **Shortcut** tab and specify the required arguments in the **File** (or **Object**) text box.

Disadvantage. If you specify, as one of the command line arguments, a string that does not start with a Latin letter, then an error will occur during program execution.

Correction. In the DInfo method, before the statement

```
DiveInfo di = new DriveInfo(d);
```

add the following fragment:

```
if (d[0] < 'A' || d[0] > 'Z')
  return;
```

Result. Now the program does not process parameters that do not start with a Latin letter.

Remark. Another possible way to catch such an error is to explicitly handle the resulting *exception* in a *try-catch block* (see Chapter 3). However, if it is possible to correct the error without involving an exception handling mechanism, then this opportunity should be used, since exception handling is *very slow*.

Comment

To determine the number of command line arguments, just use the Length property of the args array. To iterate over all the arguments, we used a foreach loop, whose variable d gets the value of the next element of the args array at each iteration. Note that you *cannot modify* array elements using a foreach loop.

3. Exception handling: EXCEP project

The EXCEP project introduces techniques for *handling exceptions*. The structure of *try blocks* is described and features related to the use of *nested* try blocks are demonstrated. An overview of the exceptions associated with arithmetic operations is provided. The checked and unchecked statements are discussed. The Parse method of converting a string to a number and the throw statement of throwing an exception (including repeated) are considered. A version of a try block using a finally clause is described.

3.1. Handling a specific exception and exception groups

This example, like the previous one, is related with a console application. Create a template project for the console application (see Section 2.1) and change the description of the Program class in the **Program.cs** file as follows:

```
class Program
{
  static void M1(int x, int y, int z)
  {
    try
    {
      int a = checked((int)Math.Pow(x, y));
      Console.WriteLine("x ^ y / z = {0}", a / z);
    }
    catch (DivideByZeroException)
    {
      Console.WriteLine("DivideByZero Exception");
    }
    Console.WriteLine("M1 finished");
  }
  static void M2(int x, int y, int z)
  {
    try
    {
      M1(x, y, z);
    }
    catch (ArithmeticException)
    {
      Console.WriteLine("Arithmetic Exception");
```

```
    }
    Console.WriteLine("M2 finished");
  }
  static void Main(string[] args)
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
    Console.ReadLine();
  }
}
```

Result. For three input integers x, y, z, the program calculates the expression x^y / z and handles the resulting exceptions (to exit the program, press Enter). Let us describe the various cases that may arise during program execution.

Case A. Handling valid values. The calculations are successful; no exception handlers are invoked:

```
x = 9
y = 2
z = 3
x ^ y / z = 27
M1 finished
M2 finished
```

Case B. Division by zero. The handler for the try block of the M1 method is activated, which handles an exception of DivideByZeroException type. After that, the program execution continues with the statement following the given try block:

```
x = 1
y = 1
z = 0
DivideByZero Exception
M1 finished
M2 finished
```

Case C. Integer overflow. Attempting to raise the number 10 to the power of 10 (and then convert the result to the integer type) throws an OverflowException. Since the catch clause of the try block of the M1 method does not handle OverflowException, it immediately moves to the next-level try block handler (that is, the catch clause of the try block of the M2 method). Here, the

OverflowException is handled because it is a descendant of an ArithmeticException, which is the ancestor of *all* exceptions thrown by arithmetic errors. After that, program execution continues with the statement following the try block of the M2 method:

```
x = 10
y = 10
z = 1
Arithmetic Exception
M2 finished
```

Case D. Invalid character input. After input an invalid character (for example, an asterisk *), the program execution is immediately interrupted, a return to the Visual Studio environment occurs, and the statement whose execution led to an exception is highlighted in the program code (in our case, this will be the first of the Main method statements containing a call to the Parse method). This behavior is due to the fact that the Main method does not handle the FormatException that was thrown and therefore activates the *default exception handling mode.*

In this situation, two actions are possible:

1. Immediately interrupt the execution of the program by pressing the Shift+F5 key combination or the button with the image of a red square  .
2. Continue the execution of the program, skipping the erroneous statement and possibly several next statements. To do this, click on the yellow arrow  located near the erroneous statement and drag it to the statement from which you want to continue the program execution, then press the F5 key or the button with the green triangle  . You can also execute the program step by step by clicking the  or  buttons (or pressing F11 or F10, respectively). Any of these buttons executes the current statement (that is, the statement pointed to by the yellow arrow). The difference between them is that if the current statement is a function call and the code for this function is available, then the button  (**Step Into** button) provides a jump to the beginning of this function, and the button  (**Step Over** button) immediately executes the function and moves to the next statement after call the function.

Remark. The behavior of the program described in case D corresponds to the default exception handling in **Debug** mode (that is, when the program is started with the F5 key). If the program is launched in the **Release** mode (that is, using the Ctrl+F5 key combination), then, when an exception occurs, a dialog box appears on the screen with information about the occurred exception and two buttons: **Continue** (to continue the program execution) and **Quit** (to terminate the program immediately).

Comments

1. The Parse method used in the program for the int type allows to convert the specified string to the int type (for such a conversion to be successful, the string must contain a representation of some integer, possibly padded with spaces on the left and right). A similar method is available for other numeric types, in particular, for the double type. It should be note that when converting a string to a real number, the decimal separator character is determined from the settings of the Windows operating system; therefore, in a program running in the Russian version of Windows, a *comma* must be input as the decimal separator. For more information on regional settings and how to change them, see Comment 3 in Section 6.4 and the comment in Section 7.1.

If the parameter of the Parse method cannot be converted to the specified numeric type, then a FormatException is thrown (see case D above).

2. It is impossible to throw an OverflowException when performing operations with *real* numbers (that is, numbers of double type): if the resulting number turns out to be too large, a special value double.PositiveInfinity of double type (*positive infinity*) will be returned; you can perform various actions with this value in the same way as with ordinary numbers. There are two other special real values: double.NegativeInfinity (*negative infinity*) and double.NaN (*not a number*). The standard math functions defined in the Math class can return both ordinary and special numeric values. For example, Math.Sqrt(–1) (square root of –1) will return double.NaN, and Math.Log(0) (natural logarithm of zero) will return double.NegativeInfinity.

In our program, we use the Pow function from the Math class, which allows us to perform exponentiation, and the overflow described in case C occurs *after* calculating this function when trying to convert the result to the int type (as a result, we get an *integer* overflow).

3. Integer overflow does not always lead to an exception. With the standard project settings, no exception occurs in this situation and an integer value that is too large is *truncated* by discarding the extra high bytes. If this behavior with integer overflow is undesirable (as in our case), then you can enable explicit overflow control by enclosing the "dangerous" expression in a checked directive with parentheses (see the M1 method). This directive can be used to protect not only expressions, but also a group of statements; in this situation, its syntax is as follows:

```
checked {statements}
```

However, keep in mind that if there are *method calls* among the specified statements, then there will be *no* integer overflow control inside these methods.

There is also a paired unchecked directive that disables integer overflow control. Due to disabled control, operations on integer data are performed much faster.

You can also control integer overflow at the level of the entire project by changing its settings. To do this, execute the **Project | <project name> Properties…** menu command, go to the **Build** section in the appeared tab with the project name, click the **Advanced** button, and set the **Check for arithmetic overflow/underflow** checkbox.

3.2. Handling any exception

Modify the Main method in the **Program.cs** file as follows:

```
static void Main(string[] args)
{
  try
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
  }
  catch
  {
    Console.WriteLine("Other exception");
  }
  Console.ReadLine();
}
```

The program now contains *three* nested try blocks.

Result. In any of cases A, B, C discussed in Section 3.1, the result of the program will be the same. Case D (input invalid character) will display the following text:

```
x = *
Other exception
```

After that the program will wait for Enter to be pressed to end its execution. Thus, no exception will now cause the program to terminate.

Disadvantage. The information displayed on the screen does not allow you to determine which exception occurred during the program execution.

Correction. Change the catch clause in the Main method to the following:

```
catch(Exception ex)
{
  Console.WriteLine(ex.GetType().Name + ":\n  " + ex.Message);
}
```

Result. Now, if an invalid character is input, more detailed information will be displayed:

```
x = *
FormatException:
  Input string was not in a correct format.
```

Comment

In the last version of the program, a handler was defined for the Exception, which is the *common ancestor* of all exception classes. Therefore, it is activated when some exception is thrown that was not handled in the previous try blocks. Using the ex variable of Exception type allows you to access the methods and properties of the thrown exception: the name of the exception class can be obtained using the expression ex.GetType(). Name (the GetType method is called, which returns an object of Type type, and the Name property is called for this object); a brief description of the error is available using the Message property of the ex object.

3.3. Re-throwing a handled exception

Append the catch clause for the try block of the M2 method as follows:

```
catch (ArithmeticException)
{
  Console.WriteLine("Arithmetic Exception");
  throw;
}
```

Result. In any of cases A, B, D, the result of the program will be the same. Case C (integer overflow) will display more detailed information:

```
x = 10
y = 10
z = 1
Arithmetic Exception
OverflowException:
  Arithmetic operation resulted in an overflow.
```

This is because the throw statement added to the catch clause of the M2 method *re-throws* the exception after it handling. The re-thrown exception was finally handled in the catch clause of the Main method.

Comments

1. The throw statement is also used to explicitly throw an exception in a program. For example, if an integer parameter n of some method M can only take values from 1 to nMax, then, if this condition is violated, an ArgumentOutOfRangeException should be thrown in the method M:

```
if (n < 1 || n > nMax)
  throw new ArgumentOutOfRangeException("n");
```

The used version of the constructor of the ArgumentOutOfRangeException class allows you to specify the name of the erroneous parameter in the Message property of the thrown exception. In our case, this property will contain the text: **Specified argument was out of the range of valid values. Parameter name: n**. An example of using the throw statement is also given in Section 23.1.

2. A try block can contain multiple catch clauses to handle different types of exceptions. There is also an additional try block clause named finally, which is located after all catch clauses and contains code to release previously allocated resources, close files, and perform other finishing actions. The finally clause is *always* executed, both after normal completion of statements in the try block, and when an exception is thrown, even if this exception was not handled in the previous catch clauses. Versions of a try block with the finally clause are given in Sections 21.5–21.6.

4. Events: EVENTS project

The EVENTS project introduces the basic techniques for developing events-driven applications. We demonstrate how to associate an event with a handler (in design mode and programmatically), how to disconnect a handler from an event, and how to reconnect it later. The Random class and properties of visual controls related to their size and position on the screen are considered. An overview of the structure of a Windows graphical application and its controls is given.

4.1. Connecting an event to a handler

The EVENTS project is the first graphical application discussed in the book, so we will describe the steps for its development in more detail (see also Chapter 1).

After creating a new project of the **Windows Forms Application** type, place a Button control on Form1 using the **Toolbox** window (the easiest way is to select the Button control from the **All Windows Forms** group, which contains all controls in alphabetical order). The added button will be automatically named button1.

Set the properties of Form1 and button1 (to do this, use the **Properties** window):

Properties

```
Form1: Text = Bouncing Buttons,
  StartPosition = CenterScreen
button1: Text = Close
```

Use Fig. 4.1 to adjust the form size and button position.

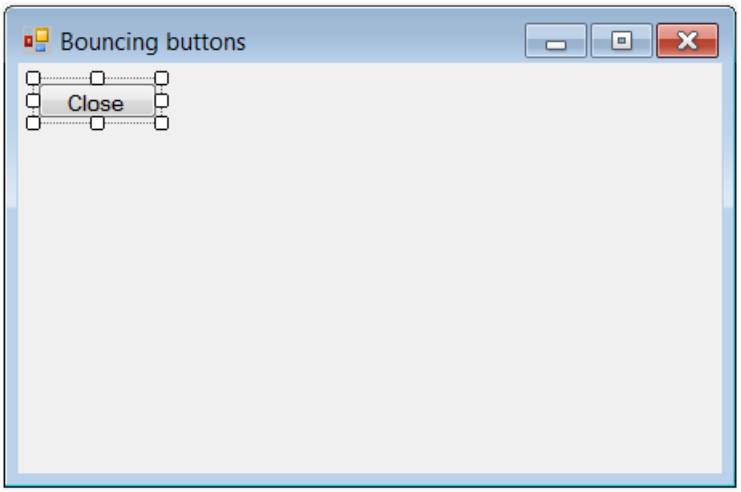


Fig. 4.1. Form1 view at the initial stage of development

Associate a handler with the Click event of the button1 control:

button1.Click handler

```
private void button1_Click(object sender, EventArgs e)
{
  Close();
}
```

To do this, select the button1 on the form, for example, by clicking on the button with the mouse (as a result, markers will be displayed around the button, as in Fig. 4.1, and the **Properties** window will be configured to display the button properties). Then select the **Events** mode in the **Properties** window (by clicking on the button ) and double-click on the empty text box to the right of the Click label. As a result, the **Form1.cs** file with the description of the Form1 class will be loaded into the editor of the Visual Studio environment, and a template for the Click event handler (the button1_Click method) will be added to this file. Now, using the editor, you need to add the necessary statements to this template (in our case, the Close method call). The automatically generated text of the button1_Click method is shown in the listing in regular font, and the program code that needs to be added to the method is shown in **bold**.

Similarly, create a MouseDown event handler for Form1:

Form1.MouseDown handler

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  button1.Location = new Point(e.X - button1.Width / 2,
    e.Y - button1.Height / 2);
}
```

Since this event should not be associated with a button, but with a form, you must first select the *form* by clicking in its free area or on its title.

Result. After starting the program, the **Bouncing buttons** window with the **Close** button appears on the screen. When you click anywhere in the window, the button "jumps" to the specified location. The CenterScreen value of the StartPosition property ensures that the window is centered on the screen. Clicking the **Close** button exits the program and returns to Visual Studio.

Comments

1. The Close function is a method of the Form1 class inherited from the Form ancestor class. Since the button1_Click event handler is also a method of the Form1 class, you do not need to prefix the Close method with the name of the form object for which this method is called.

2. In the Form1_MouseDown method, the value of the Location property of the button is changed, so you must explicitly specify the object (button1) whose property you want to change. The Location property is the Point structure consisting of two integer fields, X and Y. To change it, a new instance of the Point

type is created with fields that are defined using the X and Y fields of the parameter e (these fields contain the coordinates of the position at which the mouse button was pressed). The Width and Height properties of the button are used to center the button relative to the mouse cursor. Pay attention to the new keyword, which is required when calling constructors of structures and classes.

3. The position and size of any visual control (that is, a descendant of the Control class), including the form itself, can be determined and changed using a set of properties. The above-mentioned Location property, as well as the Left and Top properties of int type, are responsible for the position. These properties (like the X and Y fields of the Location property) determine the coordinates of the *upper-left corner* of the form control relative to the upper-left corner of the *client area* of a form (the client area does not include the title bar and frame of a form). In the case of a form, the Location, Left, and Top properties define the coordinates of the upper-left corner of the form relative to the upper-left corner of the screen.

There are also the Right and Bottom properties of int type. which determine the coordinates of the *lower-right corner* of the visual control. All coordinates are in pixels.

The Size property is responsible for size of controls; it is a structure of Size type with the Width and Height fields of int type. There are also the Width and Height properties of int type.

4. *How does the program store information about the controls placed on the form and how does it find out the values of the properties set in the* ***Properties*** *window?*

All this information is saved in a text file associated with the form designer (in our case, this file is named **Form1.Designer.cs**). Although it is usually not necessary to manually correct it, it is useful to familiarize yourself with its contents by loading this file into the editor (to do this, you just need to double-click on the file name in the **Solution Explorer** window). The **Form1.Designer.cs** file contains the part of the Form1 class description that is directly related to visual design. In particular, the end of this file contains a list of all the controls placed on the form. In our program, such a control is a button:

```
private System.Windows.Forms.Button button1;
```

Further, if we expand the hidden section, which is marked with the **Windows Form Designer generated code** text, by clicking on the + sign, we will see a piece of code containing all the property settings that we made using the **Properties** window, for example:

```
this.button1.Text = "Close";
```

Note that changing these properties directly in the text of the **Form1.Designer.cs** file will immediately affect the appearance of the form. So if you change the above statement to the following

```
this.button1.Text = "CloseWin";
```

and switch to the form design mode (that is, go to the **Form1.cs [Design]** tab), the button caption will change to **CloseWin**, and the same text will be specified for the button's Text property in the **Properties** window.

Thus, all actions associated with placing controls on the form and setting their properties can be described in the program code. Visual tools such as the **Toolbox** and **Properties** windows only speed up this process and make it more intuitive.

5. *How does the program know that the button1_Click method should be called when the button1 button is clicked and the Form1_MouseDown method should be called when the form is clicked?*

Simultaneously with the creation of a template for the button1_Click method, the name of the button1_Click method appears in the **Properties** window near the Click event (you can verify this by returning to the form design mode, highlighting the button1 button and switching to the **Events** mode in the **Properties** window by clicking the button ). In other words, the *value* of the Click event for the button1 control becomes button1_Click. In the **Form1.Designer.cs** file, the corresponding action is represented as the following statement (it should be noted that this statement contains many redundant elements – just compare it with the statements for connecting handlers, which are used further, in Section 4.3):

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

All control events displayed in the **Property** windows in **Events** mode are empty by default, that is, they are not associated with any handlers.

If some event is associated with a handler (in our case, the Click event is associated with the button1_Click method), then when the corresponding event occurs (for example, when a button is clicked), the control calls the handler method that is connected to it. In this case, the first parameter of the handler (sender) allows to determine which control has called this handler, and the second parameter (e) contains additional information about the event.

6. *Where are the statements from which the program execution begins?*

Any C# program begins by executing a start method, which by default is named Main. In the Windows Forms template, the Main method is located in the **Program.cs** file. This file is automatically generated and, like the **Form1.Designer.cs** file, usually does not require editing. If we load the **Program.cs** file into the editor, we can see that the Main method contains three statements, the most important of which is the last one:

```
Application.Run(new Form1());
```

This statement creates an instance of the application main form (of Form1 type) and starts an *event loop* that runs until the main form is closed. Closing the main form exits the event loop and exits the application.

4.2. Disconnecting a handler from an event

Add another button to the form (it will be named button2) and make its Text property empty using the **Properties** window (Fig. 4.2).

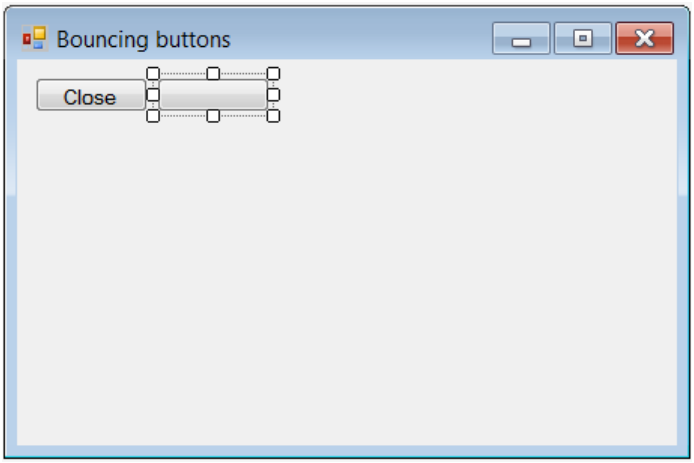


Fig. 4.2. The final form of Form1 for the EVENTS project

In the **Form1.cs** file, at the beginning of the Form1 class description (before the public Form1() constructor), add the following description of the object r:

```
private Random r = new Random();
```

Define event handlers for the MouseDown and Click events for button2:

button2.MouseMove and button2.Click handlers

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
  if (ModifierKeys == Keys.Control)
    return;
  // - if the Ctrl key is pressed,
  // then exit the handler immediately
  button2.Location =
      new Point(r.Next(ClientRectangle.Width - 5),
      r.Next(ClientRectangle.Height - 5));
}
private void button2_Click(object sender, EventArgs e)
{
  button2.Text = "Change";
  button2.MouseMove -= button2_MouseMove;
}
```

Result. A “wild” button with an empty title does not allow to click on itself running away from the mouse cursor. In order to “tame” it, you need to move the cursor to it while holding down the Ctrl key. After clicking on the wild button, it is tamed: the title **Change** appears on it and it stops running away from the mouse cursor. It should be noted that you can also tame a button using the

keyboard by selecting the button with the Tab key (or arrow keys) and pressing the spacebar.

The tamed button does nothing yet. This will be corrected in Section 4.3.

Comments

1. This section demonstrates how to disconnect a handler method from an event with which it was previously connected. To do this, use the operator -= with the required event on the left and a handler that must be disconnected from the event on the right.

2. To ensure a random movement of the "wild" button, the program uses an object r of Random type (a *random number generator*), which allows generating evenly distributed pseudo-random numbers. To create and initialize an object of Random type, you can use two versions of the constructor: without parameters and with the seed parameter of integer type. If the seed parameter is not specified, then the random number generator is initialized with a value derived from the current time (according to the computer clock). Random objects initialized with the same seed values generate the same sequence of random numbers.

To get a random number of int type, the Random class provides the Next method, which has three versions: without parameters (returns a number in the range from 0 to int.MaxValue, not including int.MaxValue), with one max parameter (returns a number in the range from 0 to max, not including max) and with two parameters min and max (returns a number in the range min to max, not including max). The button2_MouseMove handler uses a version of the Next method with one parameter. There is also the NextDouble method without parameters, which returns a random number of double type lying in the half-interval [0, 1).

3. In the button2_MouseMove handler, the ClientWidth and ClientHeight properties are used. These properties of the form determine the width and height of client area of the form (recall that the client area of the form does not include its title and frame). Subtracting the number 5 ensures that the wild button is always visible on the screen (at least partially).

4. Pay attention to how the button2_MouseMove handler checks whether the Ctrl key is pressed. As mentioned in Section 4.1, additional information about the occurred event is usually passed to the handler using the second parameter e. For example, in the button2_MouseMove handler, this parameter (of the MouseEventArgs type) allows to determine where the mouse cursor is currently located (properties e.X and e.Y of int type) and whether any mouse button is pressed (property e.Button of MouseButtons type). But the parameter e of MouseEventArgs type does *not* contain information about the currently pressed *control keys*. However, such information can be obtained using the static ModifierKeys property of the Control class, which is the base ancestor of all visual controls. Using this property, you can determine whether the Ctrl, Alt, Shift

keys are currently pressed, as well as any of their combinations. For example, you can check if the Ctrl+Shift key combination is pressed using the following condition (parentheses are required):

```
ModifierKeys == (Keys.Control | Keys.Shift)
```

5. Note that the field r in the Form1 class is not only described, but also initialized immediately (using the Random constructor without parameters). When is this initialization performed? According to the rules of the C# language, explicitly specified initialization statements for all class fields are automatically placed at the beginning of *any* class constructor. Thus, the field r will be initialized at the beginning of the Form1's constructor execution (before executing the InitializeComponent() statement specified in the constructor body). Of course, we can act differently: describe the field r of Random type without initializing it and then add to the constructor the initialization statement:

```
private Random r;
public Form1()
{
  r = new Random();
  InitializeComponent();
}
```

It should also be noted that the private access modifier (meaning that this field is *private*, that is, it is available only for methods of the Form1 class) can be omitted, because, if a class member does not have an access modifier, this member is automatically supplied with the private modifier. Nevertheless, we will always specify access modifiers, as this makes the program code more descriptive.

4.3. Connecting another handler to an event

In order for the "tamed" button to perform some actions when it is clicked, we can add the required actions to the already existing button2_Click handler. However, in this case, the handler must check whether the button is "wild" or "tamed". Let us do it differently: connect the Click event for the tamed button with *another* handler. This approach will demonstrate a number of features associated with connecting and disconnecting handlers.

Create a new handler named button2_Click2 "manually", without using the **Properties** window. To do this, add a description of the new handler at the end of the Form1 class description in the **Form1.cs** file (before the last two closing curly braces "}"):

```
private void button2_Click2(object sender, EventArgs e)
{
  if (WindowState == FormWindowState.Normal)
    WindowState = FormWindowState.Maximized;
  else
```

```
    WindowState = FormWindowState.Normal;
}
```

Notice that all the lines are in bold in this listing. This means that you need to type *all* of its text.

Also add new statements to the button2_Click method:

```
button2.Click -= button2_Click;
button2.Click += button2_Click2;
```

And add new statements to the Form1_MouseDown method:

```
if (button2.Text != "")
{
  button2.Text = "";
  button2.MouseMove += button2_MouseMove;
  button2.Click += button2_Click;
  button2.Click -= button2_Click2;
}
```

Recall that, if the place of addition is not specified, statements must be added to the *end* of the method.

Result. The tamed button now does useful work: clicking on it expands the program window to full screen, and a new click restores the window to its original state. If you click on the form (rather than any button), the **Close** button will move to the mouse position and the tamed **Change** button will become wild again, lose its title text, and start running away from the mouse (see Comment 1).

Disadvantage. During program execution, a situation may arise when one or both buttons will not be displayed on the form (if, for example, the buttons were moved to a new location when the window was maximized and then the window was returned to its original state).

Correction. Define an event handler for the SizeChanged event for Form1:

Form1.SizeChanged handler

```
private void Form1_SizeChanged(object sender, EventArgs e)
{
  if (!ClientRectangle.IntersectsWith(button1.Bounds))
    button1.Location = new Point(10, 10);
  if (!ClientRectangle.IntersectsWith(button2.Bounds))
    button2.Location = new Point(10, 40);
}
```

Result. Now, when the form is resized and its buttons are outside the client area of the form, these buttons are moved to the explicit positions near the upper-left corner of the form (see Comment 2).

Comments

1. The new parts of the button2_Click and Form1_MouseDown methods show that it is not enough to connect a new handler to an event; it is also necessary to *disconnect* the old handler from the event. *Several* handlers can be connected to the same event (for this, it is enough to apply the += operator to the event several times), although such an opportunity for events of visual controls is rarely used. When you explicitly connect handlers to an event, you must ensure that the same handler is not connected to the event multiple times, since connecting a handler multiple times usually results in hard-to-find errors. This situation can be illustrated using our program by commenting out the if statement in the Form1_MouseDown handler:

```
// if (button2.Text != "")
```

If now, after starting the program, you click on the form several times and then "tame" the button2, then, when you click this button again, the form will switch from the expanded state to the standard one and back *several times*. This is because *each* click on the form attaches a *new instance* of the button2_Click handler to the Click event of button2 and, when this button is clicked, *each* instance of the handler is executed. The situation is further complicated by the fact that it is impossible in the program to find out *how many* and *which* handlers are currently connected to the event (and without knowing this, it is impossible to ensure that all handlers are disconnected from the event). So, the explicit connection the handler to the event, as well as its subsequent disconnection, requires very careful programming.

2. To check the current position of the form controls, read-only properties of Rectangle type were used: ClientRectangle returns a rectangle that defines the client area of the form (or visual control), Bounds returns a rectangle that defines the position of the visual control on the form (or the form on the screen). The Rectangle structure has a number of properties (including Location, Size, Left, Top, Width, Height, Right, Bottom) as well as several useful methods. For example, the IntersectsWith method used in the Form1_SizeChanged method lets you check if the intersection of two rectangles is non-empty (our program examines the intersection of the ClientRectangle of Form1 with the Bounds rectangle for each of two buttons: button1 and button2).

5. Forms: WINDOWS project

The WINDOWS project introduces the specifics of applications that use multiple forms and demonstrates various ways to customize the appearance of forms and how they are displayed on the screen. Ways of interaction of different forms of one application and, in particular, the problems associated with closing non-modal subordinate forms are considered. Also we describe the settings for dialog forms and methods for displaying standard dialog boxes.

5.1. Setting the visual properties of forms. Opening forms in normal and modal mode

After creating the WINDOWS project, add two new forms to it (see Section 1.2); new forms will automatically be named Form2 and Form3. Place two buttons (with standard names button1 and button2) on Form1. Set the properties of all forms and controls (see also Fig. 5.1–5.3):

Properties

```
Form1: Text = Main Window, MaximizeBox = False,
  FormBorderStyle = FixedSingle
Form2: Text = Subordinate window,
  StartPosition = Manual, ShowInTaskbar = False
Form3: Text = Dialog window, MaximizeBox = False,
  MinimizeBox = False, FormBorderStyle = FixedDialog,
  StartPosition = CenterScreen, ShowInTaskbar = False
button1: Text = Open subordinate window
button2: Text = Open dialog window
```

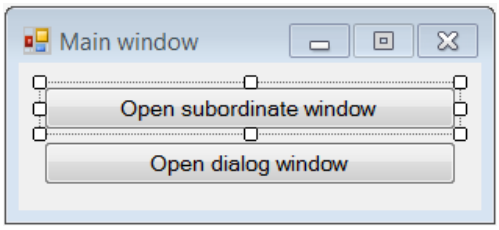


Fig. 5.1. The final view of Form1

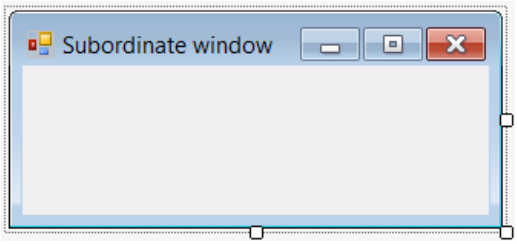


Fig. 5.2. Form2 at the initial stage of development

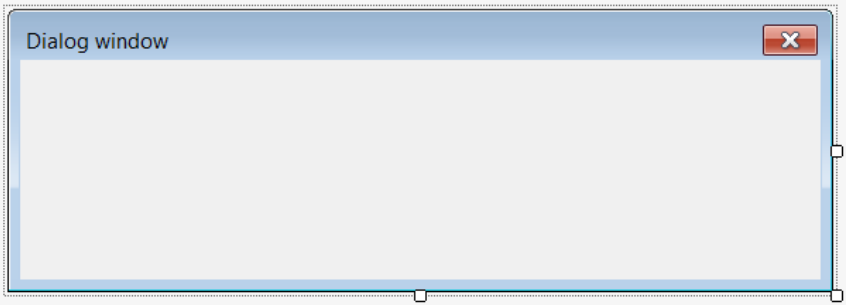


Fig. 5.3. Form3 at the initial stage of development

Add two field declarations to the beginning of the Form1 class declaration:

```
private Form2 form2 = new Form2();
private Form3 form3 = new Form3();
```

Add two statements to the constructor of the Form1 class:

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  AddOwnedForm(form3);
}
```

Define the Shown event handler for Form1 and the Click events handlers for button1 and button2:

Form1.Shown, button1.Click, button2.Click handlers

```
private void Form1_Shown(object sender, EventArgs e)
{
  form2.Location = new Point(Right - 10, Bottom - 10);
}
private void button1_Click(object sender, EventArgs e)
{
  form2.Show();
}
private void button2_Click(object sender, EventArgs e)
{
  form3.ShowDialog();
}
```

Result. The program contains three forms that demonstrate the main types of windows in Windows graphical applications: a *fixed-size window* (the Form1 class), a *variable-sized window* (the Form2 class), and a *dialog window*, or a *dialog box* (the Form3 class). There are two buttons on Form1 (see Fig. 5.1); Form2 and Form3 do not yet contain controls. Form1 is the *main form*; it is automatically created when the application is launched and immediately displayed on the screen. In addition, the main form creates two forms named form2 and form3,

which are instances of the Form2 and Form3 classes, respectively (see Comment 1).

Form2 (a *subordinate form*) is called from the main form by clicking the **Open subordinate window** button; this form is displayed in normal (*non-modal*) mode. Form3 is also a subordinate form; it is invoked by clicking the **Open dialog window** button and is displayed in *modal* (*dialog*) mode. The modal mode has the following feature: if some form of the application is in this mode, you cannot switch to other forms of the application until the modal form is closed (although it is possible to switch to other running applications). To exit the program, you need to close its main form.

The main form Form1 has fixed dimensions. The subordinate form form2 is resizable; in addition, form2 can be expanded to full screen. The visual properties of form3 correspond to the standard properties of the dialog box: form3 cannot be resized and, moreover, only the header text and a close button are displayed in its title bar (see Fig. 5.3). See also Comment 2.

The position of Form1 on the scteen is selected by the operating system, form2 is displayed near the lower-right corner of Form1 with a slight overlap; form3 is always displayed in the center of the screen (see Comment 3).

Error. After form2 closing, a new attempt to reopen it results in an exception with the following message: **Cannot access a disposed object**. This is because *closing a non-modal form destroys it*. Note that, if the form is opened in dialog mode, then its destruction on closing does not occur, you can check this by opening and closing form3 several times.

Correction. Define an event handler for the FormClosing event for the Form2 class:

Form2.FormClosing handler

```
private void Form2_FormClosing(object sender,
  FormClosingEventArgs e)
{
  if (e.CloseReason == CloseReason.UserClosing)
  {
    e.Cancel = true;
    Hide();
  }
}
```

Result. Now form2, like form3, can be reopened and closed many times during program execution (see Comment 4).

Comments

1. Calling the f1.AddOwnedForm(f2) method adds the form f2 to the list of subordinate forms of form f1. In this case, in particular, the Owner property of the form f2 becomes equal to f1 (note that, instead of the indicated method call, an assignment f2.Owner = f1 may be used). The subordinate form is always dis-

played on top of the main form, even if the main form is active. In addition, when the main form is minimized or closed, its subordinate forms are also minimized or, accordingly, closed.

2. By setting the ShowInTaskbar propertiy of a form to **False**, button for this form is not displayed on the screen taskbar. The FormBorderStyle property is responsible for the form border style, the MinimizeBox shows or hides the minimize button, MaximizeBox shows or hides the maximize button, ShowIcon shows or hides the icon in the form title bar, ControlBox allows to hide all title bar elements except the text, and HelpButton allows to display a button with a question mark (but only if both the minimize button and maximize button are hidden). The FixedDialog border style, unlike the FixedSingle style, automatically hides the icon in the form title bar.

3. The StartPosition property is responsible for the initial position of the form on the screen; this property is equal by default to the WindowsDefaultLocation value (the position of the form is determined by the operating system). To locate the form in the center of the screen, set the StartPosition property equal to CenterScreen. To explicitly determine the starting position using the Location property of the form, the StartPosition property must be set to Manual (otherwise the Location property value is ignored). If the StartPosition property is not equal to Manual, then the properties associated with the form's position (Location, Left, Top, etc.) will only get correct values when the form is first displayed on the screen. The Shown event is associated with the first form display, so we define the initial position of the subordinate form form2 in the Shown event handler for the main form Form1, when the position of the main form on the screen is already known (see the Form1_Shown method).

4. The FormClosing event belongs to a group of events that occur *before* the execution of an action and allow it to be canceled. The second parameter (e) for handlers of such events has a mutable Cancel field, which should be set to true if you want to cancel the corresponding action. The Form2_FormClosing handler cancels the closing of form2; instead, it is simply remove this form from the screen by the Hide method (a similar result can be achieved by setting the value of its Visible property to false). The condition specified in the handler allows you to determine what led to the attempt to close the form. This condition will be true when an attempt to close the form is made by any of the methods available to the user of the program or when the Close method is called explicitly. At the same time, if the Close method is automatically called at the moment of closing the main form, this condition will be false, which will allow to close the subordinate form when the application terminates.

5.2. Checking the state of the subordinate form

Modify the button1_Click method as follows:

```
private void button1_Click(object sender, EventArgs e)
{
  form2.Visible = !form2.Visible;
}
```

Define an event handler for the VisibleChanged event for Form2:

Form2.VisibleChanged handler

```
private void Form2_VisibleChanged(object sender, EventArgs e)
{
  Owner.Controls["button1"].Text = Visible ?
    "Close subordinate window" : "Open subordinate window";
}
```

Result. Now the text of button1 and the actions when it is clicked depend on whether the subordinate form form2 is displayed on the screen or not: if the subordinate form is visible on the screen, then it disappears; if it is not visible on the screen, then it appears. Note that the subordinate form can be closed not only with button1, but also in any standard way accepted in Windows (for example, using the Alt+F4 key combination); *any* method of closing the subordinate form will change the title of button1.

Comments

1. While the main form can simply refer to the subordinate form by its name form2, the subordinate form cannot do this, since the name of the main form is unknown to it. The main form of a Windows Forms application does not have a name at all, since this form is created by the constructor call in the parameter of the Application.Run method (see Comment 6 in Section 4.1). However, the subordinate form can refer to the main form using the Owner property. Moreover, using the Controls collection property of ControlCollection type, the subordinate form can access *all the controls* of its owner. The elements of the Controls collection can be indexed either using numbers or using string keys – control names. So, in the Form2_VisibleChanged method, we could specify the number 1 instead of the string key button1, since the form controls are numbered in the order *opposite* to their placement on the form: in our case, button2 (last placed on the form) has an index 0, and button1 has an index 1. Note that a consequence of this way of numbering controls is that placing a new control on the form changes the indices of *all* controls previously placed on the form. For this reason, rather than using numeric indices, it is preferable to use string keys corresponding to the names of the required controls. We will return to questions related to the order of placing controls on the form in the MOUSE project (see Section 9.1).

2. In the Form2_VisibleChanged method, a *ternary operator* is used:

```
condition ? expression1 : expression2
```

If the condition is true, then expression1 is evaluated and returned; if the condition is false, then expression2 is evaluated and returned. We emphasize that

when performing a ternary operator, only the expression whose value will be returned is evaluated. We used the ternary operator because it results in more compact code than its equivalent form with the full if–then–else conditional statement:

```
if (Visible)
  Owner.Controls["button1"].Text = "Close subordinate window";
else
  Owner.Controls["button1"].Text = "Open subordinate window";
```

5.3. Controls adapting to fit the window

Place the label1 on Form2 and set its properties (when the Dock and TextAlign properties are set, graphic selection boxes appear on the screen; in each of these box, you need to click on the central rectangular element):

Properties

```
label1: AutoSize = False, Dock = Fill, TextAlign = MiddleCenter
```

As a result, Form2 will change its appearance, as shown in Fig. 5.4.

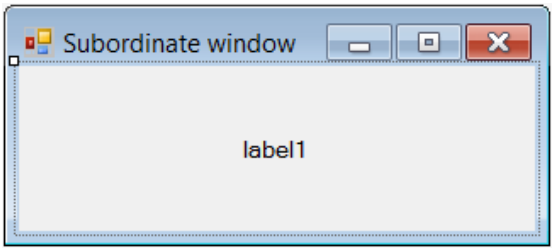


Fig. 5.4. The final view of Form2

Add the declaration of the count field to the beginning of the Form2 class declaration:

```
private int count;
```

Add new statements to the Form2_VisibleChanged method:

```
if (Visible)
  label1.Text = "Number of window openings: " + (++count);
```

Result. When the subordinate window Form2 is resized, the label1 on it is resized so that it occupies the client area of the window. The label text contains information about how many times the subordinate window has been opened.

Comments

1. When using the *increment operator* of the form ++i (a *prefix* version of the operator), the value of the variable i is firstly increased by 1 and then this variable is used in the expression. For the *postfix* operator i++, the actions are performed in the reverse order: first, the initial value of i is used in the expression and then this value is increased by 1. The prefix and postfix versions of the *decrement operator* behave in the same way.

2. Note that you do not need to call the ToString method to convert the numeric value ++count to its string representation, because, according to C# rules,

if one of the operands of the + operator is a string, then the ToString method is automatically called for the other operand. Note that, when forming strings from several elements, you can also use the Format method of the string class or the interpolated strings instead of the + operator (see Comments 4 and 7 in Section 2.2 and Comment 1 in Section 9.1).

5.4. Modal and non-modal buttons of the dialog window

Place two labels (label1 and label2), two text boxes (textBox1 and textBox2), and two buttons (button1 and button2) on Form3. Set the properties of the added controls, as well as the properties of the Form3:

Properties

```
Form3: AcceptButton = button1, CancelButton = button3
label1: Text = Main window title:
label2: Text = Subordinate window title:
textBox1: Text = Main window
textBox2: Text = Subordinate window
button1: Text = OK, DialogResult = OK
button2: Text = Apply
button3: Text = Cancel
```

When setting the relative position of controls on the form (see Fig. 5.5), you should start with the text boxes and then align the associated labels to these text boxes.

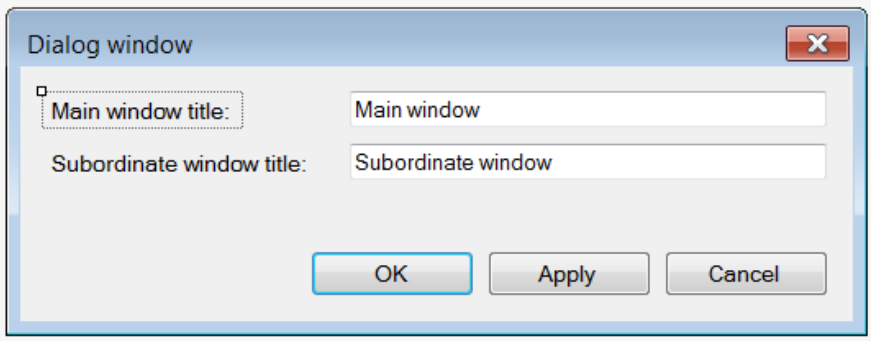


Fig. 5.5. The final view of Form3

Define the Click event handler for button2 located on Form3:

button2.Click handler (for Form3 button)

```
internal void button2_Click(object sender, EventArgs e)
// access modifier changed to internal
{
  Owner.Text = textBox1.Text;
  Owner.OwnedForms[0].Text = textBox2.Text;
}
```

Define the Click event handler for button2 located on Form1:

button2.Click handler (for Form1 button)

```
private void button2_Click(object sender, EventArgs e)
{
  if (form3.ShowDialog() == DialogResult.OK)
  // - checking which button closed the dialog window
    form3.button2_Click(this, EventArgs.Empty);
}
```

Result. The Form3 dialog window allows you to change the titles of the main and subordinate windows. The window titles are changed either by clicking the non-modal **Apply** button or by clicking the modal **OK** button (in the latter case, the dialog window is closed). The window is also closed when you click the modal **Cancel** button; in this case, the window titles are not changed. Instead of the **OK** button, you can press the Enter key; instead of the **Cancel** button, you can press the Esc key.

Comments

1. If you want the dialog to close when the button is clicked, you must define the button as *modal* by setting its DialogResult property to a value other than None (the default value). Note that, when you set the CancelButton property for a form, the button specified in this property (in our case, button3) *automatically* receives the DialogResult value equal to Cancel. The form itself also has a DialogResult property; if the form is open in dialog mode, then setting its DialogResult property to a value other than None immediately closes the form, and the resulting DialogResult value is returned by the ShowDialog function that displayed the form on the screen. When a modal button is clicked, the value of its DialogResult property is assigned to the form property of the same name. When opening a form in non-modal mode, the above mechanism does not work.

2. To access the Text property of Form2 from Form3, we use the fact that these forms have a common owner, which stores the list of its subordinate forms (in the order of their connection) in the OwnedForms property of Form[] type. Unlike the Controls collection property (see Section 5.2), the OwnedForms property, being an ordinary array, allows only integer indexing.

3. An explicit call of the button2_Click method of the Form2 class in the button2_Click handler of the Form1 class provides the execution of the actions associated with clicking the **Apply** button (thus, this call "simulates" clicking the button). When calling this method, the value named this (that is, the Form1 instance itself) is traditionally specified as the first parameter, and the EventArgs.Empty value is specified as the second parameter. Instead, you could have specified the constant null twice, since the parameters of the button2_Click method of the Form2 class are not used. In our book, we will use null instead of EventArgs.Empty to reduce the size of program code.

To be able to call the button2_Click method of the Form2 class from the Form1 class, the *access modifier* for this method must be changed to internal (the internal modifier provides free access to the class member within the created project). It would be possible to specify the public modifier, which provides free access to a class member from *any* project. This is usually done when developing projects that are *class libraries*; as a result of compiling such projects, not executable files are created, but files with the .**dll** extension.

5.5. Setting the active form control

Define a handler for the VisibleChanged event for Form3:

Form3.VisibleChanged handler

```
private void Form3_VisibleChanged(object sender, EventArgs e)
{
  if (Visible)
    ActiveControl = textBox1;
}
```

Result. No matter which control of the dialog window was active when it was closed, the next time the window is opened, the textBox1 is always active. Thus, the dialog window is always displayed *in the same initial state*. It is desirable to provide such behavior for all dialog windows.

5.6. Request for confirmation of closing the form

Modify the Form2_FormClosing method as follows:

```
private void Form2_FormClosing(object sender,
  FormClosingEventArgs e)
{
  if (e.CloseReason == CloseReason.UserClosing)
  {
    e.Cancel = true;
    if (MessageBox.Show("Close subordinate window?",
      "Confirmation", MessageBoxButtons.YesNo,
      MessageBoxIcon.Question,
      MessageBoxDefaultButton.Button2) == DialogResult.Yes)
        Hide();
  }
}
```

Result. Before closing the subordinate window form2 in one of the ways provided in the Windows system, a confirmation request for closing is displayed in the standard **Confirmation** dialog box (Fig. 5.6). If you select No (the **Нет** button, which is the default), the window closing action will be canceled. When you close the main window, the open subordinate window is closed without prompting.

Remark 1. Since the program was launched in the Windows operating system with Russian localization, the dialog box uses Russian titles for standard buttons: **Да** (Yes), **Нет** (No), **Отмена** (Cancel), etc.

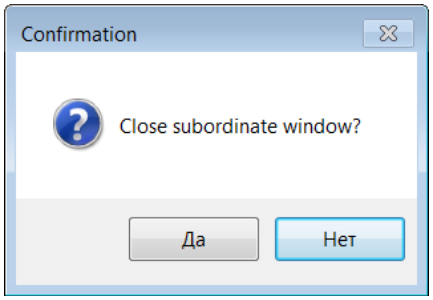


Fig. 5.6. Confirmation dialog box

Disadvantage 1. When you select **Да** (Yes) in the dialog box, the subordinate window is closed, but the main window does not become active.

This is due to the fact that the owner of the MessageBox dialog box is the form that was currently active on the screen (in our case, the active form is form2), and this form should be activated when the MessageBox dialog box is closed. However, if you select Yes, form2 is closed and therefore cannot be activated. In such a situation, no window on the screen will be active.

Correction. In the Form2_FormClosing method, replace the Hide() statement with the following block statement:

```
{
  Hide();
  Owner.Activate();
}
```

Remark 2. Another way to correct this disadvantege is to explicitly *specify the owner* of the MessageBox in an additional parameter, which should be the *first* in the parameter list. For example, you can use the Owner property of form2 as this parameter. In this case, if you select Yes, the main form will be successfully activated. However, the same form will be activated when the No option is selected (when the subordinate form remains on the screen), which does not seem quite natural.

Disadvantage 2. When you close the subordinate window by clicking on button1 of the main window, the confirmation request for closing is not displayed.

This happens because, when the button1_Click handler is executed, the Close method of the subordinate form is not called (the form simply changes its visibility mode); therefore, the handler associated with the Close event of the subordinate form is not executed either.

Correction. Change the button1_Click method of the Form1 class as follows:

```
private void button1_Click(object sender, EventArgs e)
{
```

```
  if (form2.Visible)
    form2.Close();
  else
    form2.Show();
}
```

Comments

1. In the Form2_FormClosing method, we use the version of the MessageBox.Show method, which allows to specify the request text, the title of the dialog box, a set of buttons for this window, an icon in the window, and a default button. Any parameter other than the first can be omitted; in this case, all parameters following it must be omitted too. If the second parameter is absent, then the window title is empty; if the third parameter is absent, then the only **OK** button is displayed in the window; if the fourth parameter is absent, then the icon is not displayed in the window; if the fifth parameter is absent, then the default button is the first one.

2. There is also a method in the .NET library that allows you to display a dialog box for input string information: this is the InputBox method of the Interaction class. Keep in mind, however, that the Interaction class is defined in the Microsoft.VisualBasic namespace, and the corresponding library is not automatically linked to C# projects. To link this library, do the following: right-click on the **References** item in the **Solution Explorer** window, select the **Add Reference...** command from the context menu, select the **Assemblies** group in the **Reference Manager** window that appears, then *select the checkbox* near the **Microsoft.VisualBasic** item in the list of all assemblies, and finally click **OK**. To be able to use the short name of the Interaction class (without specifying its namespace), you should add the following directive at the beginning of the cs-file:

```
using Microsoft.VisualBasic;
```

The InputBox method has five required parameters: Prompt (prompt string), Title (title string), DefaultResponse (default response string), XPos, and YPos (screen coordinates of the upper-left corner of the window). To center the dialog box horizontally and/or vertically, the corresponding parameter (XPos and/or YPos) must be set equal to –1. The method returns the input string if the dialog box was closed with the **OK** button or an empty string if the cancel button was used to close the dialog box.

As for standard dialog boxes, in the dialog box created by the Interaction.InputBox method, the language of the operating system is used for the titles of the buttons (for example, for the Russian version of Windows, the title **Отмена** is used for the cancel button); the only exception is the text **OK**.

6. Sharing event handlers and working with keyboard: CALC project

The CALC project introduces a technique for the connection of an event handler to multiple controls. It also demonstrates how to use the TryParse method to handle input errors and discusses various options for speeding up the keyboard use (default buttons, hot keys, and using the KeyPress event).

6.1. Event handler for multiple controls

After creating the CALC project, place two text boxes, two labels, and five buttons on Form1. In order to reduce the size of the first four buttons in the same way (see Fig. 6.1), after placing these buttons on the form, select them (for example, enclosing them with a dotted frame) and resize *one of them*; the size of the remaining selected buttons will change automatically.

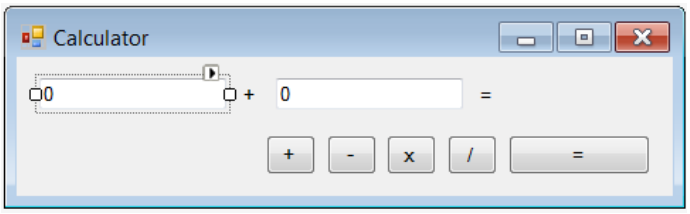


Fig. 6.1. The view of Form1 at the initial stage of development

Set the properties of the form and all added controls as follows:

Properties

```
Form1: Text = Calculator, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
textBox1: Text = 0
textBox2: Text = 0
label1: Text = +
label2: Text = =
button1: Text = +
button2: Text = -
button3: Text = x
button4: Text = /
button5: Text = =
```

Define the Click event handler for button1:

button1.Click event handler

```
private void button1_Click(object sender, EventArgs e)
{
```

```
  label1.Text = (sender as Button).Text;
}
```

After definition of the button1_Click handler, connect it to the Click event of button2, button3, and button4. To do this, select the **Events** mode in the **Properties** window and, for each of these buttons, go to the line with the Click event and *select* the name of the button1_Click handler from the drop-down list (*you should not double-click on the text box!*). Note that you can make connection with *all* three buttons at once; to do this, you must first select all these buttons on the form.

Result. Pressing any button with the operation sign (+, –, x, /) causes this operation to be displayed in label1 between textBox1 and textBox2.

Let us emphasize that *one* common handler was defined for *all four buttons*. This is possible due to the use of the sender parameter in the button1_Click handler method; this parameter contains a reference to the control that invoked the handler. The as operator converts the sender parameter (of object type) to the Button type (if this is not done, a compilation error will occur, since the object class does not have the Text property).

6.2. Calculations with control of the correctness of the input data

Define the Click event handler for button5:

button5.Click handler

```
private void button5_Click(object sender, EventArgs e)
{
  double x = 0,
    x1 = double.Parse(textBox1.Text),
    x2 = double.Parse(textBox2.Text);
  switch (label1.Text[0])
  {
    case '+':
      x = x1 + x2; break;
    case '-':
      x = x1 - x2; break;
    case 'x':
      x = x1 * x2; break;
    case '/':
      x = x1 / x2; break;
  }
  label2.Text = "= " + x;
}
```

Result. When the = button is clicked, the specified expression is evaluated and displayed on the screen (in label2). Note that the number 0 can be specified as the second operand for the division operation: when divided by 0, the result is **–Infinity** or **Infinity** (depending on the sign of the first nonzero operand). If both operands are 0, then the result of division is **NaN** (not a number). Special values of double type are discussed in Comment 2, Section 3.1. See also Comment 1.

Disadvantage. If one of the text boxes does not contain text or this text cannot be converted to a number (for example, **abc**), then clicking the = button throws an exception. If the program is launched in the **Debug** mode (by pressing the F5 key), then its execution will be interrupted, and the statement that caused the error will be highlighted in the editor of the Visual Studio environment. Possible actions in this situation are described in detail in Section 3.1.

Correction. Change the button5_Click method as follows:

```
private void button5_Click(object sender, EventArgs e)
{
  double x = 0, x1, x2;
  if (!double.TryParse(textBox1.Text, out x1) ||
    !double.TryParse(textBox2.Text, out x2))
  {
    label2.Text = "= ERROR";
    return;
  }
  switch (label1.Text[0])
  {
    case '+':
      x = x1 + x2; break;
    case '-':
      x = x1 - x2; break;
    case 'x':
      x = x1 * x2; break;
    case '/':
      x = x1 / x2; break;
  }
  label2.Text = "= " + x;
}
```

Result. Now, when trying to evaluate an expression with invalid operands, the text **ERROR** is displayed in label2; this does not interrupt the program execution, and the error message window does not appear (see Comment 2).

Comments

1. To convert strings to real numbers x1 and x2, the first version of the button5_Click method used the Parse method for the double type (see also Comment 1 in Section 3.1). To convert the resulting number x to its string representation, we do not need to call the ToString method, since the x variable is used in the expression "=" + x, which automatically performs the required conversion (see Comment 2 in Section 5.3).

2. To correct the noted disadvantage, we used the TryParse method, which, unlike the Parse method, never throws an exception. The TryParse method for the double type returns true if the string contains the correct representation of a real number and false otherwise. The result of converting the string to a real number is returned in the second, *output* parameter. Note that, in C#, output parameters must be supplied with the special out modifier when calling a method.

6.3. The simplest techniques to speed up work using keyboard

Set the properties of the form and button controls as follows (see also Fig. 6.2):

Properties

```
Form1.AcceptButton = button5
button1: Text = &+
button2: Text = &-
button3: Text = &x
button4: Text = &/
button5: Text = &=
```

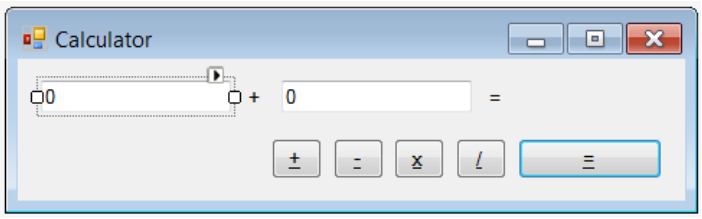


Fig. 6.2. The final view of Form1

Result. The = button (that is, button5) becomes the *default button*, the equivalent of clicking it is pressing the Enter key (the button is surrounded by a thicker blue border by default). Symbols indicated on buttons are underlined; this is an indication that a *shortcut key* Alt+*underlined character* is associated with each button. Shortcut keys are the special case of the so-called *hot keys* (or *hotkeys*), which are intended for quick performing user actions. A special feature of shortcut keys is that they are associated with some visual control (or menu item) and speed up access to this control (menu item), while a hotkey can perform an action that is not necessarily associated with any control or menu item. But often the notions “hot key” and “shortcut key” are used interchangeably.

Remark. It is possible that after starting the program, the symbols associated with the shortcut keys are *not* underlined. In this case, press the Alt key.

Error. After clicking any button with an arithmetic operation, all subsequent calculations return the value 0 (since the first character of label1 is now &, which is not provided in the switch statement).

Correction. Change the statement in the button1_Click method as follows:

```
label1.Text = (sender as Button).Text[1].ToString();
```

This correction causes only the *second* character of the button title (that is, the character with index 1) to be copied into label1. Since C# does not explicitly convert a character expression to the string type, the resulting character must be converted to string using the ToString method.

6.4. Using a keyboard event handler

Using the **Properties** window, set the KeyPreview property of Form1 to **True**. Define an event handler for the KeyPress event for Form1:

Form1.KeyPress handler

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
  char c = e.KeyChar;
  switch (c)
  {
    case '+':
      button1_Click(button1, null); break;
    case '_':
      button1_Click(button2, null); break;
    case 'x':
    case '*':
      button1_Click(button3, null); break;
    case '/':
      button1_Click(button4, null); break;
  }
  e.Handled = ! (char.IsDigit(c) || c == '.' ||
    c == '-' || c == '\b');
}
```

Result. To select any operation, you can press the corresponding key (since the "–" key is used when input negative numbers, the combination Shift+"– " corresponding to the *underscore character* is selected as an accelerator for button2). When input numbers, all keys are ignored except for numbers, "–", ".", and Backspace. You can use the \b escape sequence in C# to denote the character generated by the Backspace key; pressing this key deletes the character on the left from the cursor in the active text box. See also Comments 1–2.

Disadvantage. The Form1_KeyPress handler assumes that the decimal separator is a *decimal point*, while other decimal separators may be used in some locales on the Windows operating system (for example, *comma* is used as the decimal separator for Windows with Russian localization).

Correction. Change the last statement of the Form1_KeyPress method as follows:

```
e.Handled = ! (char.IsDigit(c) || c == '.' ||
  c == Application.CurrentCulture
    .NumberFormat.NumberDecimalSeparator[0] ||
  c == '-' || c == '\b');
```

Result. The decimal separator now corresponds to the current Windows regional settings. To test this feature, you just need to temporarily change the regional settings in the **Regional and Language Options** section of the **Windows Control Panel**. See also Comment 3.

Comments

1. In order for keyboard events to be processed by the form first, set the form's KeyPreview property to **True**. If this is not done, then the keyboard event is immediately processed by the active control; thus, this event does not reach the form and, accordingly, it does not activate the form's keyboard handler.

2. If the e.Handled value is set to true in the keyboard handler, then the current keyboard event will be considered as handled and will not be passed to other active controls. So, in our case, the form intercepts and processes *all* characters, except for numeric characters, decimal separator, "minus" sign, and \b.

3. The CurrentCulture property of the Application object allows to get information about the locale settings used by the program, in particular, about number formats. This property has the CultureInfo type defined in the System.Globalization namespace. By default, the program uses the regional settings of the operating system. The index [0] must be specified because the NumberDecimalSeparator property has the string type, which is not assignment compatible with the character type. The NumberDecimalSeparator property is read-only, but you can change the CurrentCulture property as a whole (see the comment in Section 7.1).

6.5. Control over changes to the input data

Modify the button1_Click method:

```
private void button1_Click(object sender, EventArgs e)
{
  label1.Text = (sender as Button).Text[1].ToString();
  label2.Text = "=";
}
```

Define the TextChanged event handler for the textBox1 control:

textBox1.TextChanged handler

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
  label2.Text = "=";
}
```

Connect the resulting textBox1_TextChanged handler to the TextChanged event of the textBox2 control.

Result. If you change the arithmetic operation or the content of text boxes, the result of the previous calculation is erased. This is an important feature that prevents *inconsistencies* in the displayed data. In its absence, a situation is possible when, for example, after performing calculations of the form 3 + 2 (with the result 5), the user will change the first operand to 2, receiving the text **2 + 2 = 5** on the screen.

7. Working with date and time: CLOCK project

The CLOCK project focuses on classes related to date and time (the DateTime, TimeSpan classes and the non-visual Timer control). Two options for implementing a stopwatch are considered; actions are described that ensure the display of the clock and stopwatch on the taskbar when the application window is minimized.

7.1. Displaying the current time on the form

After creating the CLOCK project, add label1 to Form1, as well as a non-visual control of Timer type (this control will be named timer1 and will be placed below the form, in the area of non-visual controls). Set the properties of the form and the added controls as follows:

Properties

```
Form1: Text = Clock, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
label1: Text = 00:00:00 AM, AutoSize = True,
  TextAlign = MiddleCenter, BorderStyle = Fixed3D,
  Font.Name = Arial, Font.Size = 60
timer1: Enabled = True, Interval = 1000
```

When setting the properties of label1, notice its Font property, which also has a set of properties, two of which, Name and Size, need to be changed (in the listing above these properties use dot notation: Font.Name and Font.Size). Adjust the position of label1 in accordance with Fig. 7.1.

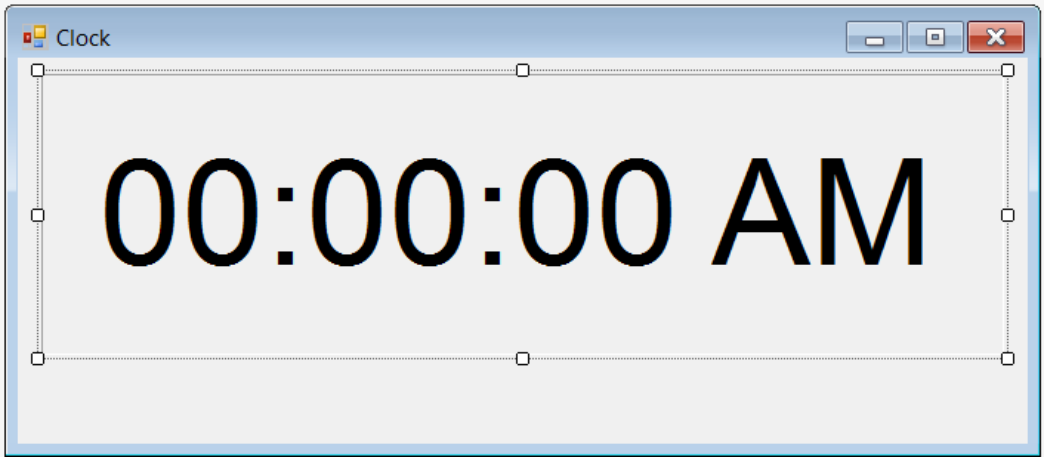


Fig. 7.1. Form1 view at the initial stage of development

Define an event handler for the Tick event for the timer1 control:

timer1.Tick handler

```
private void timer1_Tick(object sender, EventArgs e)
{
  label1.Text = DateTime.Now.ToLongTimeString();
}
```

Result. When the program is running, the current time is displayed in its window.

Disadvantage. During the first second after starting the program, the original text is displayed in the program window, since the Tick event occurs for the first time only after the timer1.Interval time interval, which is 1000 in our case (the time is specified in milliseconds).

Correction. Connect the timer1_Click handler to the Load event of Form1.

Result. Now the timer1_Click method is executed for the first time just before the form is displayed on the screen, so the correct time is immediately displayed in the window.

Comment

The .NET Framework class library provides a DateTime structure for working with date and time. Its static read-only property named Now returns the current date and time (based on the computer's system clock). Only the current date (time corresponds to midnight) can be obtained using the Today static property. The following methods of the DateTime structure can be used to convert date/time to their standard string representations:

- ToShortDateString – date in short format (d), for example, 01/27/1756;
- ToLongDateString – date in long format (D), Tuesday, January 27, 1756;
- ToShortTimeString – time in short format (t), 2:55 AM;
- ToLongTimeString – time in long format (T), 2:55:15 AM.

The ToString method without parameters returns the date/time in G format (date in short format, time in long format). The date/time display format can be explicitly specified in the ToString method. For example, in our program, we could use this option: DateTime.Now.ToString("T").

Let us mention some more date/time formats: g – date and time in short format, F – date and time in long format, f – date in long format, time in short format, M or m – format "month, day", Y or y – format "month, year".

When formatting dates, the current locale is used (in our case, we use the settings for US English), although there is an overloaded version of the ToString method in which you can explicitly specify the required locale. You can also change all regional settings for the application; it's enough to set a new value for the Application.CurrentCulture property (we used this property earlier in Section 6.4 to get information about the current regional settings). For example, you can use the following statement to set the Russian regional settings for an application:

```
    Application.CurrentCulture =
      new System.Globalization.CultureInfo("ru-RU");
```

Note that the settings for US English are named en-US.

7.2. Implementation of the stopwatch capabilities

Place a checkbox control of CheckBox type (it will be named checkBox1) and two buttons (button1 and button2) on Form1 and set their properties as follows:

Properties

```
checkBox1: Text = Stop&watch
button1: Text = &Start/Stop, Enabled = False
button2: Text = &Reset, Enabled = False
```

Adjust the position of the added controls in accordance with Fig. 7.2.

Fig. 7.2. The final view of Form1 for the CLOCK project

Add the declaration of the field named t to the Form1 class declaration:

```
private int t;
```

Add new statements to the top of the timer1_Tick method:

```
private void timer1_Tick(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    t++;
    label1.Text = string.Format("Time: {0}:{1}", t / 10, t % 10);
  }
  else
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Define the CheckedChanged event handler for checkBox1 and Click event handlers for button1 and button2:

checkBox1.CheckedChanged, button1.Click, button2.Click handlers

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
```

```
{
  if (checkBox1.Checked)
  {
    t = -1;
    timer1.Interval = 100;
  }
  else
    timer1.Interval = 1000;
  timer1_Tick(this, null);
  button1.Enabled = button2.Enabled = checkBox1.Checked;
  timer1.Enabled = true;
}
private void button1_Click(object sender, EventArgs e)
{
  timer1.Enabled = !timer1.Enabled;
}
private void button2_Click(object sender, EventArgs e)
{
  timer1.Enabled = false;
  t = 0;
  label1.Text = "Time: 0:0";
}
```

Result. When the **Stopwatch** checkbox is set to the "on" state, the program switches to the stopwatch mode and the stopwatch starts immediately displaying seconds and tenths of seconds on the screen. The stopwatch may be started and stopped by clicking the **Start/Stop** button, the stopwatch may be reset by clicking the **Reset** button. Hot keys are available: Alt+W (change the clock/stopwatch mode), Alt+S (start/stop the stopwatch), Alt+R (reset the stopwatch).

Error. The stopwatch shows not quite correct data. We can verify this by not stopping the stopwatch for some time (while performing other actions on the computer) and then comparing the result with the exact time. The reason is that the Tick event occurs *approximately* every 100 ms; in addition, the Tick event occurs only when there are *no* other events that need to be processed by the program. If the program executes some method for a long time, then the stopwatch value will not be updated during this time and then its work will continue from the previous value. For the correct implementation of the stopwatch, we need to bind it to the *computer clock* (using the Now method).

Correction. Change the Form1 class declaration:

```
~~private int t;~~
private DateTime startTime, pauseTime;
private TimeSpan pauseSpan;
```

The purpose of the added fields is as follows: the startTime field contains the start time of the stopwatch; the pauseTime field contains the time of the last stop of the stopwatch, the pauseSpan field contains the total duration of all stops made since the initial start.

Modify the timer1_Tick, checkBox1_CheckedChanged, button1_Click, and button2_Click methods:

```
private void timer1_Tick(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    t++;
    label1.Text = string.Format("Time: {0}:{1}", t / 10, t % 10);
    TimeSpan s = DateTime.Now - startTime - pauseSpan;
    label1.Text = string.Format("Time: {0}:{1}",
      s.Minutes * 60 + s.Seconds, s.Milliseconds / 100);
  }
  else
    label1.Text = DateTime.Now.ToLongTimeString();
}
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    t = -1;
    timer1.Interval = 100;
    startTime = DateTime.Now;
    pauseSpan = TimeSpan.Zero;
  }
  else
    timer1.Interval = 1000;
  timer1_Tick(this, null);
  button1.Enabled = button2.Enabled = checkBox1.Checked;
  timer1.Enabled = true;
}
private void button1_Click(object sender, EventArgs e)
{
  timer1.Enabled = !timer1.Enabled;
  if (timer1.Enabled)
    pauseSpan += DateTime.Now - pauseTime;
  else
    pauseTime = DateTime.Now;
```

```
}
private void button2_Click(object sender, EventArgs e)
{
  timer1.Enabled = false;
  t = 0;
  label1.Text = "Time: 0:0";
  pauseTime = startTime;
  pauseSpan = TimeSpan.Zero;
}
```

Comment

The TimeSpan structure is used to store *relative* time intervals. Time intervals are measured in days, hours, minutes, seconds, and milliseconds; to get the value of each of these controls, you can use the corresponding properties of the TimeSpan structure: Day, Hour, Minute, Second, Millisecond (note that the same properties are also available for the DateTime structure; in addition, this structure has properties Year and Month). The easiest way to set the time span to zero is to use the read-only TimeSpan.Zero field. Both DateTime and TimeSpan structures also have read-only fields that define their smallest and largest possible values: MinValue and MaxValue.

Addition and subtraction operations are defined for DateTime and TimeSpan structures as follows:

- the sum or difference of TimeSpan values is of TimeSpan type;
- the sum or difference of DateTime and TimeSpan values (in that order) is of DateTime type;
- the difference between values of the DateTime type is of TimeSpan type;
- you *cannot* add two DateTime values.

Since negative values are allowed for time intervals, a unary minus operator is also defined for the TimeSpan structure.

The easiest way to create DateTime and TimeSpan objects with the required values is to use one of the provided constructors. The parameterless constructor returns the minimum date for DateTime (midnight on January 1, 1 A.D.), and a zero time interval for TimeSpan.

In other versionss of the DateTime constructor, you must specify the year, month, day (and you can additionally specify the time in hours, minutes, and seconds). The time specified in the DateTime constructor can be complemented with the number of milliseconds.

TimeSpan constructors need to specify the hour, minute, and second; as an additional *initial* parameter, you can specify the number of days. If the TimeSpan constructor specifies a number of days, then you can specify an optional *final* parameter, the number of milliseconds.

7.3. Alternative options for executing commands using the mouse

Define an event handler for the MouseDown event for label1:

label1.MouseDown handler

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Clicks == 2)
    checkBox1.Checked = !checkBox1.Checked;
  else
  {
    if (!button1.Enabled)
      return;
    if (e.Button == MouseButtons.Left)
      button1_Click(this, null);
    else
      if (e.Button == MouseButtons.Right)
        button2_Click(this, null);
  }
}
```

Result. Double-clicking any mouse button on label1 changes the clock/stopwatch mode, single left-clicking in stopwatch mode starts or stops the stopwatch, single right-clicking resets the stopwatch.

Comments

1. We can combine the actions for *single* and *double* mouse clicks in one handler due to the presence of the Clicks property in the parameter e (of MouseEventArgs type), which can take the value 1 or 2.

2. When associating some actions with single and double clicks, it should be taken into account that, when performing a double click, the system first registers a single click (at which the MouseDown event occurs with e.Clicks equal to 1, then the Click event occurs, then MouseUp event occurs with e.Clicks equal to 1), and only then, after the second mouse click, if the time interval between clicks was short enough, a double click is registered (at which MouseDown event occurs with e.Clicks equal to 2, then DoubleClick, then MouseUp with e.Clicks equal to 2). Therefore, it is very important that the action performed on a single click *does not conflict* with the action associated with a double click. In our program everything is fine: although actions are provided for both single and double-click in the stopwatch mode, the action with a single click (start, or stop, or reset the stopwatch) does not in any way conflict with the action associated with a double-click (changing the clock/stopwatch mode).

7.4. Displaying the current status of the clock and stopwatch on the taskbar

Add a new statement to timer1_Tick method:

```
Text = WindowState == FormWindowState.Minimized ?
  label1.Text : "Clock";
```

Result. If you minimize the CLOCK application window, then its button located on the Windows taskbar will display, depending on the mode, the current time or stopwatch data. If the application window is in its normal state, then the application button displays the **Clock** text that coincides with the window title.

Disadvantage. If you minimize the window while the stopwatch is stopped, the text of the application button will not change.

Correction. Define the Resize event handler for Form1:

Form1.Resize handler

```
private void Form1_Resize(object sender, EventArgs e)
{
  Text = WindowState == FormWindowState.Minimized ?
    label1.Text : "Clock";
}
```

Result. Now the text on the application button located on the Windows taskbar is correctly adjusted in any situation, since the Resize event occurs not only when the form is resized, but also when it is minimized and returned to its normal state.

8. Text input: TEXTBOXES project

The TEXTBOXES project demonstrates the features of controls for text input (TextBox controls). We also consider issues related to control activation, as well as ways to handle erroneous data at the level of a separate text box and the form as a whole. A mechanism for generating error messages based on the use of the ErrorProvider control is described.

8.1. Additional highlighting of the active text box

After creating the TEXTBOXES project, place 12 text boxes (textBox1 – textBox12) on Form1 and set the properties of the form and the added controls:

Properties

```
Form1: Text = TextBoxes, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
textBox1-textBox12: Text = Data
```

Text boxes should be placed on the form row by row from left to right: textBox1 – textBox3 in the first row, textBox4 – textBox6 in the second row, etc. (see Fig. 8.1).

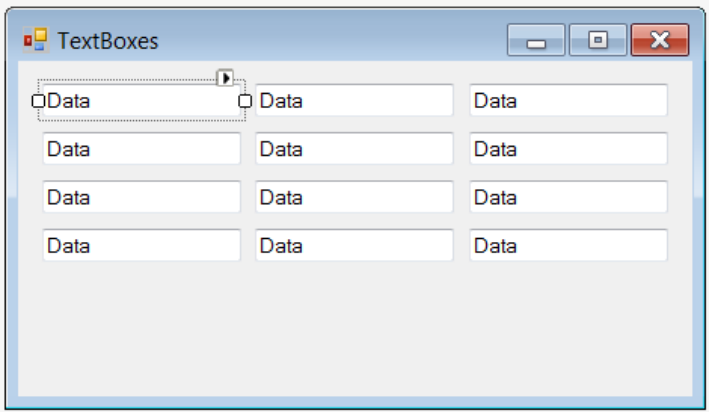


Fig. 8.1. Form1 view at the initial stage of development

To assign the same **Data** value to the Text property of all text boxes, it is enough to select all controls and then set this value using the **Properties** window.

Define the Enter and Leave event handlers for textBox1:

textBox1.Enter and textBox1.Leave handlers

```
private void textBox1_Enter(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = Color.White;
```

```
  tb.BackColor = Color.Green;
}
private void textBox1_Leave(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = SystemColors.WindowText;
  tb.BackColor = SystemColors.Window;
}
```

Connect the created handlers to the Enter and Leave events of all other text boxes (see Section 6.1 for how to connect handlers to multiple controls). To connect the created handlers simultaneously to all the remaining text boxes (textBox2 – textBox12), you must first select *all* of them.

Result. When any text box receives focus (that is, when the text box is activated – see Comment 1), its background and foreground colors change; if focus is lost, the initial color setting is restored.

Disadvantage. When receiving focus, the text of the text box is highlighted (as a rule, it is displayed in white on a blue background); thus, the left-hand part of the text box (highlighted characters) is colored blue and the right-hand one is green, which looks bad.

Correction. Add new statements to the constructor of the Form1 class:

```
public Form1()
{
  InitializeComponent();
  for (int i = 1; i <= 12; i++)
  {
    TextBox tb = Controls["textBox" + i] as TextBox;
    tb.Select(tb.Text.Length, 0);
  }
}
```

Result. Now, when receiving focus, the text in the text box is *not* selected, and the keyboard cursor (a *caret*), which looks like a vertical line, is located after the last character of the text (see Comment 2).

Remark. To iterate over all the text boxes placed on the form, the Controls collection property of the form was used (see Comment 1 in Section 5.2).

Comments

1. Moving focus is provided by clicking on the control or using the Tab and Shift+Tab keys. When using the Tab and Shift+Tab keys, the order of moving focus (named *tab order*) of focusable controls is determined by the value of their TabIndex property. By default, the tab order is the same as the order of adding controls to the form. The easiest way to view and change the current tab order is to use the **View | Tab Order** menu command in form design mode.

When this command is executed, a number equal to the value of the TabIndex property is displayed near each form control. To set a new tab order, it is enough to click all controls in the required order (the numbers near the controls will be changed). To exit the tab order setting mode, just press the Esc key. Note that the tab order can also be changed programmatically (see Section 8.2). The TabStop property is also associated with the tab order: if the value of the TabStop property of some visual control is **False**, then this control is excluded from the tab order.

It is often sufficient to use the arrow keys to move focus, but this is not possible for text boxes because they handle the arrow keys in a special way.

2. Many properties and methods are provided in the text boxes to set and change the *selected text*. The above Form1 constructor uses the Select method, which has two parameters: the position of the *start* of the selection (numbered from zero, which corresponds to the position number *before* the first character) and the *length* of the selection, that is, the number of characters selected. If the selection length is 0, then the start position of the selection determines the caret position. The SelectionStart and SelectionLenght properties are also provided for the start of the selection and its length, respectively; these properties are mutable. For example, the statement tb.SelectionStart = tb.Text.Length can be used instead of the last statement of the Form1 constructor (the SelectionLength property can be left unchanged).

Let us also mention an important property named SelectedText, which allows to get and change the selected text. Assigning a new string to the SelectedText property causes the selected fragment to be replaced with the specified string (if the text box did not contain a selection, then the specified string is inserted at the caret position). Changing the SelectedText property hides the selection and places the caret behind the inserted text fragment (thus, after setting the SelectedText property to any value, this property will return an *empty string*). If you assign an empty string to the SelectedText property, then the previously selected fragment will be deleted.

Another property related with selection is the HideSelection boolean property. If it is set to **True** (which is the default value), then, when a text box field loses focus, the selected text fragment will no longer be displayed in a different color (however, when the focus is received again, the selection color is restored). If the HideSelection property is set to **False**, then the appearance of the selection does not change when the focus is lost (this mode is usually used in text editors when performing actions to find and replace text fragments).

8.2. Changing the tab order of text boxes

Place the groupBox1 container control (of GroupBox type) on Form1. After that, place two radio buttons in the created container control: radioButton1 and

radioButton2. Set the position of the added controls (Fig. 8.2), as well as their properties:

Properties

```
groupBox1: Text = Tab Direction
radioButton1: Text = &Rows, Checked = True
radioButton2: Text = &Columns
```

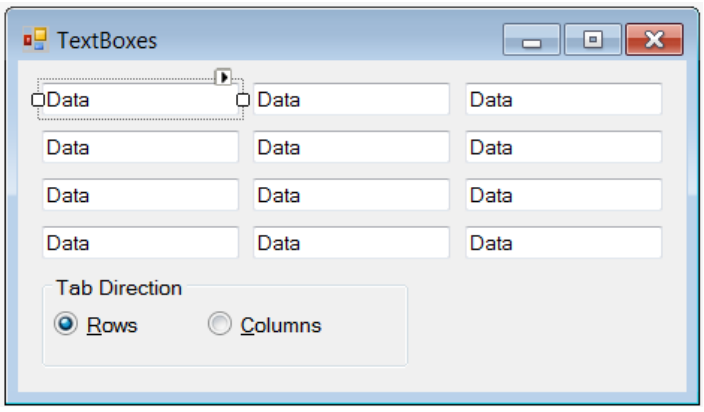


Fig. 8.2. Form1 view at an intermediate stage of development

Define an event handler for the CheckedChanged event for radioButton1, then connect the created handler to the CheckedChanged event of radioButton2.

radioButton1.CheckedChanged handler

```
private void radioButton1_CheckedChanged(object sender,
  EventArgs e)
{
  if (!(sender as RadioButton).Checked)
    return;
  if (sender == radioButton1)
    for (int i = 0; i <= 11; i++)
      Controls["textBox" + (i + 1)].TabIndex = i;
  else
    for (int i = 0; i <= 3; i++)
      for (int j = 0; j <= 2; j++)
        Controls["textBox" + (3*i + j + 1)].TabIndex = i + 4*j;
}
```

Result. Using the radio buttons added to the form, you can change the tab order of text boxes: the fields can now be selected either by rows (with the **Rows** radio button selected), or by columns (with the **Columns** radio button selected). You can also switch the tab order using the Alt+R and Alt+C key combinations.

Disadvantage. With any of the implemented methods of changing the tab order, the current text box loses focus (since one of the radio buttons receives focus).

Correction. Change the Text properties of the radio buttons as follows: **&Rows (F2)** for radioButton1 and **&Columns (F3)** for radioButton2, set Form1's KeyPreview property to **True**, and define the KeyDown event handler for Form1:

Form1.KeyDown handler

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  switch (e.KeyCode)
  {
    case Keys.F2:
      radioButton1.Checked = true; break;
    case Keys.F3:
      radioButton2.Checked = true; break;
  }
}
```

Result. Now you can just press the F2 key to set the tab order by rows and press F3 to set the tab order by columns. The focus remains on the previously activated text box.

Comment

When a new radio button is selected, the CheckedChanged event handler is executed twice: for the previously selected radio button, whose Checked property has changed from true to false, and for the newly selected radio button, whose Checked property has changed from false to true. The first conditional statement of the radioButton1_CheckedChanged handler causes an immediate exit if the handler is called on a radio button that has lost its selection. The second conditional statement allows to determine which radio button has become selected. Note that the CheckedChanged event occurs not only when the mouse is clicked on one of the radio buttons, but also when the Checked property is programmatically changed (this is the difference between the CheckedChanged event of a radio button and its Click event, which occurs only as a result of user actions).

8.3. Blocking exit from an empty text box

Define a handler for the Validating event for the textBox1 and then connect the created handler to the Validating events of all other text boxes.

textBox1.Validating handler

```
private void textBox1_Validating(object sender, CancelEventArgs e)
{
  e.Cancel = (sender as TextBox).Text.Trim() == "";
}
```

Result. If the active text box is empty, then it is impossible to exit it (in particular, it is impossible to close the form). Note that we can still select radio

buttons using the F2 and F3 keys, since this feature is not associated with loss of a focus for the active text box.

Disadvantage. The reason the focus is being locked may not be clear to the user. This disadvantage will be corrected in the next section.

Comments

1. The Validating event occurs before the control loses focus; in the handler of this event, the loss of focus can be blocked by setting the Cancel property of the parameter e to true (compare this with the actions of the FormClosing event handler discussed in Section 5.1).

2. The Trim method of the string class removes all leading and trailing spaces and returns the modified string. There are also TrimStart and TrimEnd methods that remove only leading or only trailing spaces, respectively. Using the Trim, TrimStart, and TrimEnd methods, you can remove not only spaces, but also any other characters; for this, it is enough to specify the characters to be removed as parameters of these methods (the number of parameters can be arbitrary).

8.4. Informing the user about the error

Add a non-visual ErrorProvider control to the form (it will be named errorProvider1) and set its BlinkStyle property to NeverBlink.

Define a handler for the TextChanged event for the textBox1 and then connect the created handler to the TextChanged events of all other text boxes.

textBox1.TextChanged handler

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  if (tb.Text == "")
    errorProvider1.SetError(tb, "Text must be non-empty");
  else
    if (errorProvider1.GetError(tb) != "")
      errorProvider1.SetError(tb, "");
}
```

Result. If you delete all characters in the active text box, an icon of a red circle with an exclamation mark (that is, a sign of an error) will appear to the right of this text box. If you move your mouse cursor over this icon, a tooltip appears with a brief explanation of the cause of the error. If you input at least one character in an empty text box, the icon disappears.

Comment

By using a single instance of the ErrorProvider control, you can inform the user about errors associated with various visual controls. You can change the icon displayed on the screen in case of an error (the Icon property), as well as set one of the blinking modes of this icon using the BlinkStyle property (NeverBlink – no blinking, AlwaysBlink – constant blinking, BlinkIfDifferentError –

short blinking when the icon is displayed on the screen and also when the text with the error message changes for this control). The blink rate (in milliseconds) can be adjusted using the BlinkRate property.

8.5. Providing additional information about the error

The brief error message may be confusing for some users. In such a situation, it is desirable to provide for the possibility of calling the *help system*, but this call, as a rule, is performed using the **Help** button, while exit from the erroneous text box is blocked. The CausesValidation property is provided to solve this problem.

Place a button (named button1) on the form, set its Text property to **Help** (Fig, 8.3), set its CausesValidation property to **False**, and define the Click event handler for the button:

button1.Click handler

```
private void button1_Click(object sender, EventArgs e)
{
  MessageBox.Show("The text in the text box must not be empty",
    "Help");
}
```

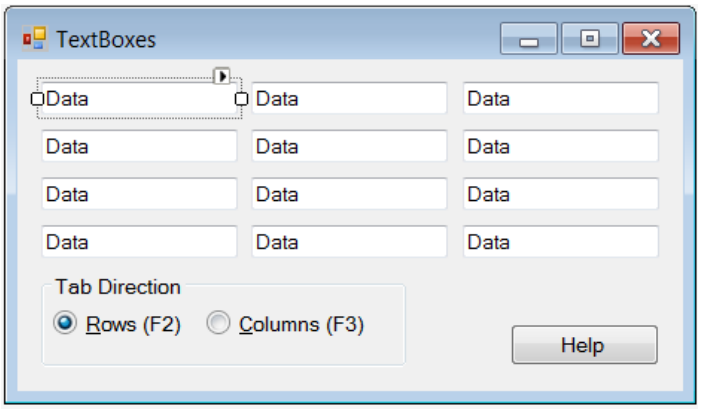


Fig. 8.3. The final view of Form1

Result. The **Help** button is available for click in a situation when one of the text boxes is blocked. However, in this situation, you still cannot navigate to other form controls (you can only return to the text box marked as erroneous).

Remark. For all visual controls, the CausesValidation property defaults to **True**. It is recommended to change it to **False** only for buttons related to the help call.

8.6. Form-level error checking

Blocking an erroneous text box can be too inconvenient for users who prefer to fill in text boxes that are not problematic first and then return to those text boxes that require additional thought. To make this method of filling data available, you should use blocking actions that are performed *at the form level*, that

is, at the level of the *entire dataset* (examples of such actions are saving the entire dataset in a file or sending this dataset over the network). Let us implement such blocking actions for our project.

Select all text box controls and *clear* the Validating event for them in the **Properties** window (you can also remove the textBox1_Validating method from the Form1.cs file).

Define the FormClosing event handler for Form1:

Form1.FormClosing handler

```
private void Form1_FormClosing(object sender,
  FormClosingEventArgs e)
{
  if (e.CloseReason != CloseReason.UserClosing)
    return;
  for (int i = 1; i <= 12; i++)
    if (errorProvider1.GetError(Controls["textBox" + i]) != "")
    {
      e.Cancel = true;
      return;
    }
}
```

Result. Now the presence of an empty text box does not prevent the activation of the other text box, however, an error icon is displayed near to each empty text box. If there is at least one error icon, the form cannot be closed.

Remark. The first conditional statement in the Form1_FormClosing handler gives the opportunity to close the form if the corresponding command comes not from the user, but from the operating system (for example, the form must be closed when Windows exits). See also Comment 4 in Section 5.1.

9. Mouse event handling: MOUSE project

The MOUSE project is primarily devoted to mouse event handling. The mouse capture mechanism and the problems caused by such a capture are considered. We discuss parent-child relationships and the ordering of child controls in a form or other container control (*z*-order). Also we describe the find and replace tools of Visual Studio environment.

9.1. Dragging with the mouse. Setting the z-order of controls on a form

After creating the MOUSE project, place two Panel controls on Form1 (they will be named panel1 and panel2) and set the properties of the form and the added controls (see also Fig. 9.1):

Properties

```
Form1: Text = Mouse, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
panel1: BackColor = Red, BorderStyle = Fixed3D
panel2: BackColor = Green, BorderStyle = Fixed3D
```

Fig. 9.1. Form1 view at the initial stage of development

Add the field declaration to the beginning of the Form1 class declaration:

```
private Point p;
```

Define event handlers for the MouseDown and MouseMove events for panel1, then connect the created handlers to the MouseDown and MouseMove events of panel2 (see Section 6.1 for how to connect handlers to multiple controls).

panel1.MouseDown and panel1.MouseMove handlers

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
```

```
{
  p = e.Location;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
}
```

Define an event handler for the MouseMove event for Form1:

Form1.MouseMove handler

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  Text = "Mouse";
}
```

Result. After starting the application, moving the mouse cursor over any panel causes the name of the panel and the current values of the local coordinates of the mouse relative to the panel to be displayed in the window title, in addition to the **Mouse** text. If the left mouse button is held down, the panel is moved along the form ("dragged" by the mouse). See also Comments 1–3.

Disadvantage. When panel1 dragging, it may be overlapped by panel2.

Correction. Add two statements to the panel1_MouseDown method:

```
Panel a = sender as Panel;
a.BringToFront();
```

Result. Now the dragged panel is always positioned on top of all window controls (see Comment 4).

Remark. The required result can be obtained using a single statement (sender as Panel).BringToFront(). We declare an auxiliary object a of Panel type, since, in the subsequent sections, the panel1_MouseDown method will be supplemented with other statements using the object a.

Comments

1. To update the form title when dragging the panel (see panel1_MouseMove method), the Format method of the string class is used, which returns a string that contains both plain text and formatted representations of various objects. The first parameter of the Format method is a *format string* containing plain text and *format settings* for other parameters (the number of formatted parameters can be arbitrary). Format settings are enclosed in curly braces {}. In our case, we use the simplest format settings, in which only the ordinal number of the parameter displayed in the specified position of the format string is set (in

such a simple situation, the ToString method is automatically called to format the parameter). Note that the ToString method of the Point object annotates the X and Y coordinates with **X =** and **Y =** comments and encloses the resulting text in curly braces, for example, **{X = 50, Y = 15}**. For format settings, see also Comment 4 in Section 2.2.

2. In the panel1_MouseMove method, when calculating the new position of the panel, an *offset* s0 of Size type is added to its previous position Location of Point type (offset s0 is created using the constructor of the Size structure). Objects of Size type can not only be added, but also subtracted from an object of Point type; the result is a new object of Point type. You cannot add and subtract objects of Point type. Objects of Size type can be added and subtracted; the result is an object of Size type. Note that the Point and Size types can be converted to each other using an explicit cast. Given this fact, the position of the panel can be recalculated as follows:

```
a.Location += (Size)e.Location - (Size)p;
```

3. The apparent constancy of the coordinate values displayed on the screen when dragging the panel is explained by the fact that, immediately after changing the mouse coordinates, the position of the panel on the form is adjusted, and the local mouse coordinates relative to the panel are recalculated at the same time. If you move the panel quickly enough, you will notice that other coordinate values are displayed for a short time in the form title bar.

4. Any visual control has the BringToFront method, which "raises" this control above all controls, and the SendToBack method, which "lowers" this control under all controls. These actions involve changing the *z-order* of the controls, that is, their relative position on the *z*-axis oriented perpendicular to the plane of the screen. The concept of *z*-order is closely related to the *parent-child relationship* and makes sense for children of the same parent, so the parent-child relationship should be discussed first.

As a rule, the *parent* of the controls is the form, but any container control can act as a parent (such controls are indicated in the **Containers** group of the **Toolbox** window). Further in this comment, we will use the variable p to designate a parent, that is, a form or a container control.

All *child* controls are contained in the Controls collection property of their parent p. The Controls property is of ControlCollection type and allows you to refer to any child control either by an integer *index* starting from zero (for example, p.Controls[0]) or by a string *key* that matches the name of the child control (that is, the value of its Name property, for example, p.Controls["button1"]). To add a control to the Controls collection, you can use its Add method (for example, p.Controls.Add(button1)). You can also set the control's Parent property to p (for example, button1.Parent = p). These ways of adding a control to the Controls collection are completely equivalent. The new control is added to the *end* of the Controls collection (the starting element of the collection is the element with in-

dex 0). The size of the Controls collection can be determined by using its read-only Count property.

The *position* of the child is relative to the upper-left corner of the parent's client area (that is, this position is specified in the *parent's local coordinate system*). Child controls cannot be displayed outside the visible area of their parent (although they can be placed outside the visible area – see Section 9.2).

Note that a form usually has no parent; in this case, its Parent property is null and its position is in screen (that is, *global*) coordinates. The exception is *child forms* in an MDI application (see Chapter 22). In addition, the form can have an *owner* – see Sections 5.1, 5.4, 5.6.

Let us go back to the concept of *z-order*. It is convenient to assume that the *z*-axis is directed away from the user, that is, into the screen (in this case, we get the *right-handed* coordinate system *xyz*, since the *y*-axis is directed vertically downward on the screen). With this assumption, you can treat the child controls' indices in the Controls collection as their *z*-coordinates. In other words, the first item in the Controls collection (with index 0) is located on the *z*-axis at point 0, the second control (with index 1) at point 1, and so on. We emphasize that the *z*-axis is directed away from the user, so the first child control will appear as the topmost control and may overlap the second and other child controls.

This statement seemingly contradicts the result that we get when adding a control to a form in design mode, because the control that is placed *last* on the form is the *topmost* control. The answer is that adding a control to a form in design mode and adding a control to the Controls collection leads to different results. You can verify this by opening the **Form1.Designer.cs** file and viewing the text of the InitializeComponent method (you may first need to expand the **Form1.Designer.cs** file fragment marked as **Windows Form Designer generated code**). It turns out that the Controls.Add methods are called on controls *in the reverse order* of being placed on the form. In other words, when placing a control on a form in design mode, this control is located not at the end, but *at the beginning* of the Controls collection. This fact explains the apparent contradiction. A clear understanding of this fact allows you to avoid a number of errors associated with placing controls on a form and accessing them using the Controls collection. Let us point out two of the most typical errors of this kind.

• *Do not assume that the first control placed on a form in design mode will have index 0 in the* Controls *collection.* On the contrary, at every stage of program development, index 0 will be associated with the *last* control added to the form in design mode. That is why it is more convenient to access the elements of the Controls collection not by numeric indices, but by string keys such as "button1".

• *Do not assume that a control added to a form not in design mode, but programmatically (that is, by explicitly assigning a form object to its* Parent

property or by calling the Add *method of the form's* Controls *collection) will appear on the form as the topmost control.* On the contrary, it will be placed *at the end* of the Controls collection and therefore will be *below* all other child controls, that is, it will be the last in *z*-order. For a control to become the topmost control, after adding it to the Controls collection, you should explicitly call the BringToFront method for it.

A clear understanding of the *z*-order features of **Windows Forms** library also allows you to understand the features of *control docking* (that is, binding of the control to the boundaries of the form performed by setting the Dock property), but we will defer this discussion until another example (see Section 21.3).

9.2. Resizing with the mouse

At the beginning of the Form1 class declaration, add a declaration of the new field:

```
private Size s;
```

Add new statements to the panel1_MouseDown and panel1_MouseMove methods:

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
  else if (e.Button == MouseButtons.Right)
    a.Size = s + s0;
}
```

Result. If you move the mouse cursor over the panel while holding down the *right* mouse button, the panel is *resized* (the left button is still used to change the *position* of the panel). Note that you can even drag the panel to an area outside the window. If you do not release the mouse button at the same time, the panel can be returned back to the visible part of the window (similarly, after reducing the panel's size to zero, you can then restore them). The noted feature is related to the mechanism of *mouse capture*: if you press any mouse button over

a control, this control will “capture” the mouse and “force” mouse to execute mouse event handlers of this control (even if the mouse cursor leaves the control) until the mouse button will not be released (however, this event will also be handled by the control that previously captured the mouse). See also Comment 1.

Disadvantage. If you drag the panel entirely outside the window and *release* the mouse button, then access to the panel will be impossible. The panel will also become inaccessible when its size is reduced to zero, provided that you release the mouse button at that moment.

Correction. Set the AutoScroll property of Form1 to **True** and change the panel1_MouseMove method as follows:

```
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
  {
    a.Location += s0;
    Point p0 = a.Location + s0;
    a.Location = new Point(Math.Max(0, p0.X), Math.Max(0, p0.Y));
  }
  else if (e.Button == MouseButtons.Right)
  {
    a.Size = s + s0;
    s0 += s;
    a.Size = new Size(Math.Max(50, s0.Width),
      Math.Max(20, s0.Height));
  }
}
```

Result. Now it is impossible to drag the panel beyond the left or top border of the window, and, in addition, the minimum panel size (in pixels) is set equal to 50 × 20. Dragging the panel outside the right or bottom border of the window does not make it inaccessible, since, in such a situation, the window displays *scroll bars* (due to the **True** value for the AutoScroll property of the form). See also Comments 2 and 3.

Comments

1. The mouse capture effect can be demonstrated in our program in one more way: if you press the mouse button on a free part of the form and then move the mouse cursor to one of the panels, the form title will not change (it will still look like **Mouse**). This is because, in this situation, the mouse *is cap-*

tured by the form itself, and even when the mouse is moved over the panel, the MouseMove event is sent to the form that captured the mouse. If we release the mouse button over the panel, then any mouse movement will be "intercepted" by the panel and lead to changing the form title.

2. To correct the disadvantage, we limited the size of the control by adding the appropriate code to the program (see the new version of the panel1_MouseMove method). The other way of correction is to restrict the size of a control using its MinimumSize and MaximumSize properties of Size type. In our case, it is enough, for panels panel1 and panel2, to set the MinumumSize property equal to **50; 20** using the **Properties** window. However, this way of correction is less convenient because it requires setting the MinumumSize property for *each* control. In addition, there are no properties limiting the *position* of the control.

3. The Max function used in the new version of the panel1_MouseMove method returns the maximum of two numeric parameters. This function is a static method of the Math class. All other standard math functions are also implemented as static methods of the Math class.

9.3. Using additional cursors

Add new statements to the panel1_MouseDown method:

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (e.Button == MouseButtons.Left)
    a.Cursor = Cursors.Hand;
  else if (e.Button == MouseButtons.Right)
    a.Cursor = Cursors.SizeNWSE;
}
```

Define a handler for the MouseUp event for panel1, then connect the created handler to the MouseUp event of panel2:

panel1.MouseUp handler

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  (sender as Panel).Cursor = null;
}
```

Result. In the mode of resizing the panel, the mouse cursor becomes a diagonal double-headed arrow; in the drag mode, it becomes a "pointing hand". After exiting these modes, the standard cursor is restored.

Remark. The Cursor property of any visual control, as well as the Cursors class, were discussed in detail in Chapter 11. Recall that, if the Cursor property is

null, the control will use the cursor of its parent (in this case, the cursor of the form).

9.4. Handling a situation with simultaneous pressing of several mouse buttons

Our program will work fine as long as the user does not think to press the left mouse button while holding the right button down (or vice versa). For definiteness, we will describe the behavior of the program in a situation when, while the left mouse button was pressed on the panel, the right mouse button was additionally pressed (and, in addition, we will assume that when performing the actions described below, the mouse cursor will not leave the panel on which it was initially located). At the moment of pressing the second button, the cursor will change to a diagonal arrow; however, with the subsequent movement of the mouse, neither the position nor the size of the panel will change. If you release one of the mouse buttons, the cursor will take the form of a *standard arrow* (the default cursor); however, when you move the mouse, the action determined by the mouse button that remains pressed will be performed.

This behavior is explained by the following features of the mouse event handlers: when the handlers for the MouseDown and MouseUp events are executed, the e.Button parameter contains information about the mouse button that was *just* pressed (or, respectively, released). The e.Button parameter *does not contain* information about other buttons pressed. On the other hand, when the MouseMove event handler is executed, the e.Button parameter contains information about *all currently pressed mouse buttons*; the elements of the MouseButtons enumeration corresponding to the pressed buttons are combined by the operator | (bitwise OR).

This makes it possible to understand why our program does not perform any actions when the buttons are simultaneously pressed: in fact, the actions to move or resize the panel are implemented in the MouseMove event handler when the condition e.Button == MouseButtons.Left or e.Button = = MouseButtons.Right holds, but, if both mouse buttons are pressed, neither of these conditions are true, because, in this case, the e.Button property contains the expression MouseButtons.Left | MouseButtons.Right.

To avoid such undesirable effects, we should analyze in more detail the state of the mouse buttons in the handlers associated with their pressing and releasing. We will assume that the left button has a *higher priority*: if it is pressed (even simultaneously with the right button), then the panel should be dragged, not resized (and the cursor should be a “pointing hand”).

Let us make the required changes to the event handlers:

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
```

```
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (e.Button == MouseButtons.Left)
  if ((MouseButtons & MouseButtons.Left) != 0)
    a.Cursor = Cursors.Hand;
  else if (e.Button == MouseButtons.Right)
  else if ((MouseButtons & MouseButtons.Right) != 0)
    a.Cursor = Cursors.SizeNWSE;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
  if ((e.Button & MouseButtons.Left) != 0)
  {
    Point p0 = a.Location + s0;
    a.Location = new Point(Math.Max(0, p0.X), Math.Max(0, p0.Y));
  }
  else if (e.Button == MouseButtons.Right)
  else if ((e.Button & MouseButtons.Right) != 0)
  {
    s0 += s;
    a.Size = new Size(Math.Max(50, s0.Width),
      Math.Max(20, s0.Height));
  }
}
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  (sender as Panel).Cursor = null;
  Panel a = sender as Panel;
  if ((MouseButtons & MouseButtons.Left) != 0)
    a.Cursor = Cursors.Hand;
  else if ((MouseButtons & MouseButtons.Right) != 0)
    a.Cursor = Cursors.SizeNWSE;
  else
    a.Cursor = null;
}
```

Result. Pressing and releasing any mouse buttons in any order on a free part of the form does not change the cursor. When you press and release any mouse buttons in any order on the panels, the cursor view always corresponds to the correct mode: this is the *drag mode* if the *left* button is pressed (even if the right button is pressed at the same time) and this is the *resize mode* if the *right* button is pressed and the left button is released. You can even press or release the mouse wheel, which in this respect also behaves like a button – the *middle* mouse button; this will not affect the operation of the program in any way (see also Comment 1).

Disadvantage. If you press the right and left buttons on some panel and then release one of them and *move the mouse cursor outside the panel* (for example, move the cursor outside the entire form), the current mode (dragging or resizing) will stop working, and the form title changes to **Mouse**. The current mode resumes working if you move the mouse back to the panel.

This behavior of the program means that the mouse capture mechanism stops working. This is because the capture of the mouse by the current control is canceled if *at least one* of the pressed buttons is released. In our program, it would be more convenient to cancel the mouse capture only if *all* the mouse buttons are released.

Correction. Define a handler for the MouseCaptureChanged event for the form and then connect that handler to the MouseCaptureChanged event for each panel:

```
private void Form1_MouseCaptureChanged(object sender,
  EventArgs e)
{
  Control c = sender as Control;
  if (!c.Capture && MouseButtons != MouseButtons.None)
    c.Capture = true;
}
```

Result. Now the mouse capture for both the form and the panels is canceled only when all mouse buttons are released (see Comments 2 and 3).

Comments

1. To correctly determine the required mode and cursor, it is not enough to know which button was pressed or released; it is also necessary to determine which buttons *remain* pressed. To do this, we use the MouseButtons property in the modified versions of the panel1_MouseDown and panel1_MouseUp methods. MouseButtons is a static property of the Control class; it allows to determine which mouse buttons are pressed at present time (compare the MouseButtons property with the ModifierKeys property discussed in Section 4.2).

To check if the required mouse button is included in the MouseButtons or e.Button enumerations, we use the operator & (bitwise AND).

2. To change the default mouse capture mechanism, we used the Capture property of bool type. Any visual control has this property; it allows to check if a given control captures the mouse and it also allows to set or cancel mouse capture programmatically. The Capture property is associated with the MouseCaptureChanged event; this event is raised when the value of the Capture property changes.

The Form1_MouseCaptureChanged handler analyzes the situation when the control cancels the mouse capture (in this case, the value of its Capture property changes from true to false). If, at the same time, some mouse button remains pressed (that is, the value of MouseButtons is *not* equal to MouseButtons.None), then the mouse capture is restored.

3. In the Form1_MouseCaptureChanged handler, the sender parameter is cast to the Control type which is the common ancestor of all visual controls. This makes it possible to connect this handler with controls of various types (and, in particular, with the form itself).

9.5. Dragging and resizing a control of any type. Using the find and replace tool

Place a button (named button1) and a label (named label1) on Form1 and set the label properties:

Properties

```
label1: AutoSize = False, BorderStyle = FixedSingle,
  TextAlign = MiddleCenter
```

Adjust the size of the button and label in accordance with Fig. 9.2.

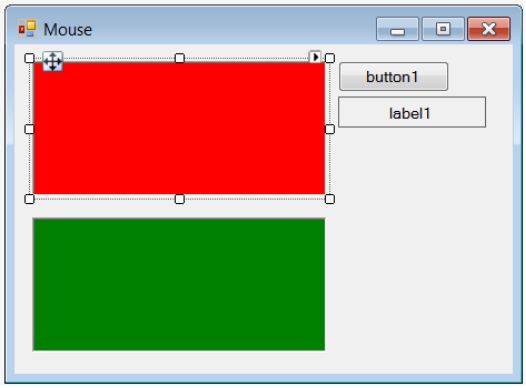


Fig. 9.2. The final view of Form1

Connect handlers panel1_MouseDown, panel1_MouseMove, panel1_MouseUp, and Form1_MouseCaptureChanged to the appropriate events (MouseDown, MouseMove, MouseUp, and MouseCaptureChanged) for button1 and label1.

In the text of the **Form1.cs** file, replace the word **Panel** with the word **Control**. To do this, go to the beginning of this file and press the Ctrl+H key combination. In the find and replace bar that appears, input the text **Panel** in the first text box, the text **Control** in the second text box, switch on the **Match whole**

word button , and click the **Replace all** button (Fig. 9.3). After all the replacements have been completed, a window will appear with information that six found words have been replaced (all these words are contained in three identical statements `Panel a = sender as Panel` located at the beginning of the `panel1_MouseUp`, `panel1_MouseDown`, and `panel1_MouseMove` methods). Press Esc to close the find and replace bar.

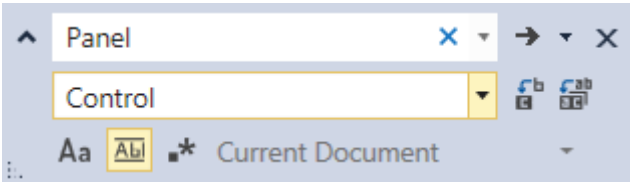


Fig. 9.3. Find and replace bar

Result. The added controls (`button1` and `label1`) are processed in the same way as panels `panel1` and `panel2`: their size and position can be changed with the mouse.

Comments

1. In the `panel1_MouseUp`, `panel1_MouseDown`, and `panel1_MouseMove` handlers, the same action is now performed as in the handler `Form1_MouseCaptureChanged`: the `sender` parameter is cast to the `Control` type. Note that these methods use only the properties and methods of the `sender` parameter that the `Control` class has (otherwise a compiler error would occur).

2. Since *find and replace operations* are often useful when viewing and editing program code, let us note some of the related capabilities.

To quickly find the required text, it is convenient to use the **Incremental Search** mode activated by the **Edit | Advanced | Incremental Search** menu item or the Ctrl+I key combination. When this mode is active, the mouse cursor changes to a downward-pointing arrow and is complemented by a binocular. Now it is enough to input the required text fragment, and, in the process of input, the first of the found text fragments located below the position of the keyboard cursor will be highlighted in the file. If the input of a text fragment is completed, then to repeat search *forward*, it is enough to press Ctrl+I again, and to repeat search *backward*, you should press Ctrl+Shift+I. To exit the **Incremental Search** mode, just press Esc. Note that, even after exiting this mode, the editor remembers the last input text fragment for search; if, after the next activation of the **Incremental Search** mode, you do not type a new text fragment but immediately press Ctrl+I or Ctrl+Shift+I, then the previously input text fragment will be searched (forward or backward, respectively).

If you need to take into account additional conditions when searching, you should display the find bar by pressing the Ctrl+F key combination. In addition to the already mentioned option to search for a *whole word* (the **Match whole word** button or Alt+W), you can also organize a *case-sensitive* search (the

Match case button Aa or Alt+C) and enable the search mode using *regular expressions* (the **Use Regular Expressions** button .* or Alt+E).

When the find bar is open, to search for the occurrence of the next fragment, just press the **Find Next** button → or the F3 or Enter key. Using the F3 key, the search can be performed even when the find bar is closed. We emphasize that pressing F3, unlike Ctrl+I, does not require input a fragment and ensures that all options specified in the find bar are taken into account when searching. You can search backward by pressing Shift+F3.

As already noted, the find and replace bar can be displayed on the screen using the combination Ctrl+H. It contains the same buttons as the find bar. In the search and replace mode, you can search for the next occurrence of the fragment (the **Find Next** button → or F3), replace the found occurrence (the **Replace Next** button or Alt+R; after performing the replacement, the next occurrence is immediately searched for) or replace *all* found occurrences (the **Replace All** button or Alt+A).

If the find bar or find and replace bar is active, just press the Esc key to close it.

10. Drag-and-drop: ZOO project

The ZOO project introduces various aspects of the *drag-and-drop mode* (activation of the drag-and-drop mode and various options for its completion, actions when dragging to an invalid target, additional selection of the source and target, use of special cursors for drag-and-drop mode). In addition, the project describes how to work with the ImageList (an image storage) control.

10.1. Dragging labels on a form

After creating the ZOO project, place four labels (label1 – label4) on Form1 and set the properties of the form and the added labels:

Properties

```
Form1: Text = Zoo, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen,
  AllowDrop = True
label1: Text = Bear
label2: Text = Wolf
label3: Text = Fox
label4: Text = Hare
```

Set the position of the labels in accordance with Fig. 10.1.

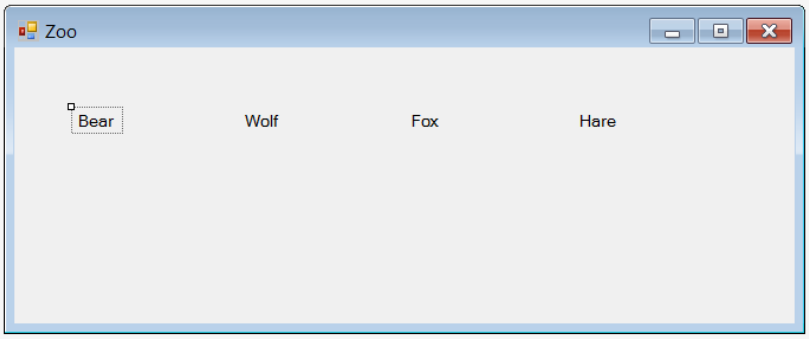


Fig. 10.1. Form1 view at the initial stage of development

Define an event handler for the MouseDown event for label1 and then connect the created handler to the MouseDown events of labels label2 – label4 (see Section 6.1 for how to connect handlers to multiple controls).

label1.MouseDown handler

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    DoDragDrop(sender, DragDropEffects.Move);
}
```

Define the DragEnter and DragDrop event handlers for Form1:

Form1.DragEnter and Form1.DragDrop handlers

```
private void Form1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void Form1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  src.Location = PointToClient(new Point(e.X, e.Y));
}
```

Result. Labels with the names of animals can be dragged using the left mouse button. When you drag the label (the *source* of the drag-and-drop), it stays in place, but the mouse cursor changes to indicate the drag-and-drop mode. Only the form itself is defined as the receiver (that is, the *target* of the drag-and-drop): when the mouse button is released over it, the label being dragged moves to the specified position (see Comments 1 and 2).

Remark. In this project, we use the src and trg names for the source and target objects, respectively, in the program code of the drag-and-drop handlers.

Disadvantage 1. It is allowed to drag a label not only to the free part of the form, but also to another label; this dragging overlaps two labels.

Correction. Set the AllowDrop property to **True** for all labels.

Result. When moving one label to another, the mouse cursor takes the form of a prohibition sign ⃠ (see Comment 3).

Disadvantage 2. At the initial moment of dragging, the cursor takes the form of a prohibition sign.

Correction. Define the DragEnter event handler for label1 and then connect the created handler to the DragEnter events of label2 – label4:

label1.DragEnter handler

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  if (e.Data.GetData(typeof(Label)) == sender)
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
```

Result. At the start of dragging, the mouse cursor has a standard view for the drag-and-drop mode. Dragging one label onto another is still prohibited (see Comment 4).

Comments

1. To start the drag-and-drop mode, any control has the DoDragDrop method with the first parameter specifying the *drag source* (that is, the object that will be accepted by the drag-and-drop receiver). The second parameter (of DragDropEffects type) is the allowed result (*effect*) of a successful drag-and-drop. There are three main effects: **Copy**, **Move**, and **Link** (their names are associated with actions when dragging *files*). If during dragging one of several effects can occur (for example, **Copy** *or* **Move**), then, as the second parameter, you must specify *all these effects*, combining them with the operator | (bitwise OR), for example, DragDropEffects.Copy | DragDropEffects.Move. There is also a DragDropEffects.All enumeration member that combines *all* the available effects.

The drop-and-drop receiver (the *target*) can only select an effect that is included in the set of allowed effects specified as the second parameter of the DoDragDrop method (in the handler of any drag-and-drop event, the target can determine which effects are allowed using the e.AllowedEffect property). The target can also set the effect to **None**, meaning that it "refuses" to accept the source. The effect selected by the target determines the appearance of its cursor in the drag-and-drop mode; in particular, if the target has selected the **None** effect, then its cursor will look like a prohibition sign ⃠.

In order for a control to act as a target, its AllowDrop property must be set to **True**. Only then will the control be able to respond to drag-and-drop events (in particular, DragEnter, DragOver, and DragDrop).

The DragEnter event occurs when the drag source appears over the target, the DragOver event occurs when the drag source is moved over the target. Within these handlers, the target must determine whether it can accept the source and set the e.Effect property to the value of the corresponding effect. If, when the source is moved over the target, the availability of the target cannot change (as in our program), then the DragOver event handler need not be defined; in such a situation, the availability of the receiver is determined by the value of the e.Effect property set in the handler for the DragEnter event. The DragDrop event occurs when the source is dropped over the target and only if the target can receive the source.

In the handler for any drag-and-drop event, you can access the source object. To do this, call the GetData method of the e.Data object and specify the source format (an expression of Type or string type) as a parameter. The result returned by the GetData method is of object type, so it must be explicitly cast to the actual type of the source. In a situation where the type of the source is not known in advance, the receiver can obtain the necessary information using the GetFormats method of the e.Data object (without parameters), which returns an array of strings containing the names of the available data formats for the source (since the source can provide data in several formats). In our case, the

call to the GetFormats method would return an array of one element: the string "System.Windows.Forms.Label", which is the full name of the source type. Note that this name may also be specified as a parameter to the GetData method instead of typeof(Label).

2. To move the label to a new location, we need to know the position of the mouse cursor where the label is "dropped". This position is contained in the e.X and e.Y properties (unfortunately, the Location composite property of Point type is not provided for the parameter e of DragEventArgs type). When working with the X and Y properties, take into account that they determine the position of the mouse cursor *in screen coordinates*. To convert screen coordinates to coordinates associated with the client area of a control or form, you can use the PointToClient method. There is also the PointToScreen method that allows you to convert the local coordinates of the control to screen coordinates.

3. By setting the form's AllowDrop property to **True**, we made the form a valid target. Since the AllowDrop property for labels remained **False** (the default value), the labels were not considered as valid targets; this means that they are *invisible* for the drag-and-drop mode. Therefore, when the mouse cursor in drag-and-drop mode passes over the label controls, it is assumed to be above the *form*, and the DragEnter and DragDrop events occur for the form. If the source is dropped at this moment, then this source will overlap the label located under it (Disadvantage 1). When the AllowDrop properties of the labels are set to **True** to correct the noted disadvantage, they became visible for the drag-and-drop mode. Since a DragEnter handler for these controls has not yet been defined, it was assumed that they *could not accept* a drag-and-drop source, so the cursor over them looks like a prohibition sign.

4. The conditional statement of the label1_DragEnter method checks if the source object (determined using the GetData method) and the target object (the sender parameter) are the same. Note that in such a checking it is not necessary to cast the specified objects to the Label type, since it is not necessary to know actual types of two objects to compare them for identity.

10.2. Dragging labels to text boxes

Place four text boxes (textBox1 – textBox4) on Form1 and adjust their position in accordance with Fig. 10.2.

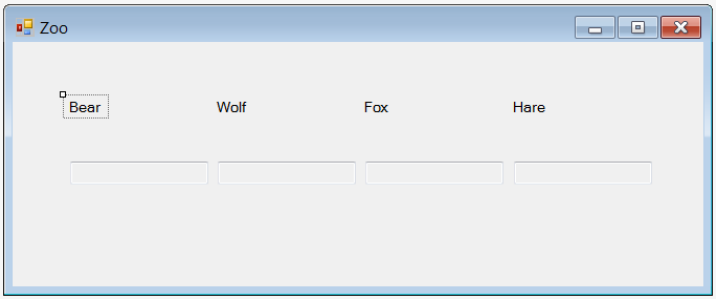


Fig. 10.2. Form1 view at the intermediate stage of development

For each text box, *clear* its Text property and set the ReadOnly and AllowDrop properties to **True** (the ReadOnly = **True** setting prevents the user from editing the text in the text box, but does not affect the ability to programmatically change the text).

Define the DragEnter and DragDrop event handlers for textBox1 and then connect the created handlers to the DragEnter and DragDrop events of textBox2 – textBox4:

textBox1.DragEnter and textBox1.DragDrop handlers

```
private void textBox1_DragEnter(object sender, DragEventArgs e)
{
  if ((sender as TextBox).Text == "")
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  (sender as TextBox).Text = src.Text;
  src.Visible = false;
}
```

Result. Any empty text box ("an empty cage") can also be a valid drag-and-drop target. When dragging a label onto an empty text box, "the animal enters the cage" (that is, the label text is displayed in the text box). Dragging and dropping a label onto an already filled text box is prohibited (although this action will become available in the next section).

10.3. Interaction of labels

Using the **Properties** window, set the Tag property values for all controls placed on the form: label1 – **3**, label2 – **2**, label3 – **1**, label4 – **0**, textBox1–textBox4 – **0**. After setting the Tag properties, load the **Form1.Designer.cs** file into the editor (just double-click on its name in the **Solution Explorer** window), expand the **Windows Form Designer generated code** section in this file, and correct all 8 statements that set the Tag property values by removing the quotes from them. For example, the text

```
this.label1.Tag = "3";
```

should be replaced with

```
this.label1.Tag = 3;
```

Modify the label1_DragEnter and textBox1_DragDrop methods:

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move
```

```
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  (sender as TextBox).Text = src.Text;
  TextBox trg = sender as TextBox;
  if ((int)src.Tag >= (int)trg.Tag)
  {
    trg.Text = src.Text;
    trg.Tag = src.Tag;
  }
  src.Visible = false;
}
```

Define the DragDrop event handler for label1, then connect the created handler to the DragDrop events of label2 – label4:

label1.DragDrop handler

```
private void label1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  Label trg = sender as Label;
  if ((int)src.Tag > (int)trg.Tag)
  {
    src.Location = trg.Location;
    trg.Visible = false;
  }
  else
    src.Visible = false;
}
```

Also connect the label1_DragEnter handler to the DragEnter events of textBox1 – textBox4. After performing this action, the textBox1_DragEnter method will no longer be associated with any event, so its description in the **Form1.cs** file can be deleted.

Result. When dragging the name of one animal onto the name of another, the stronger animal "eats" the weaker one. The same happens if one of the animals is dragged into a cage already occupied by another animal. Note that now the DragEnter events of *all* controls (both labels and text boxes) are associated with the *same* label1_DragEnter handler.

Remark. It would be possible to connect the DragEnter events of all controls to the Form1_DragEnter handler, since it performs the same actions. We used a separate handler for the controls, because later we will make some changes to the Form1_DragEnter method (see Section 10.5).

Error. If, when dragging a label, drop it over itself, the label will disappear. Thus, *the animal eats itself*.

Correction. In the label1_DragDrop method, before the statement

```
if ((int)src.Tag > (int)trg.Tag)
```

insert the following statement:

```
if (src == trg) return;
```

Comments

1. The Tag property determines the *relative strength* of the animals. When an animal is placed in a cage, information about the strength of the animal is stored in the Tag property of the cage (that is, a control of TextBox type). In order to preserve the possibility of placing an animal in an empty cage, the Tag properties of all text boxes are set equal to 0 at the beginning of the program. In our case, it is convenient to store data of integer type in the Tag properties, but when setting the Tag property using the **Properties** window, a *string* value is assigned to it. This leads to the need to adjust the **Form1.Designer.cs** file. Note that after such an adjustment, the values of the Tag properties in the **Properties** window are displayed correctly, however, any their change in the **Properties** window will leads to saving the changed Tag value as a string.

2. When defining a new value for src.Location in the label1_DragDrop method, you could have used the e.X and e.Y parameters (as in the Form1_DragDrop method), but it is easier to use the Location property of the trg variable. In addition, this way allows to place the source label exactly in the position of the target label, regardless of where the mouse button was released at the target.

10.4. Actions in case of dragging to invalid target

Modify the label1_MouseDown method:

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    if (DoDragDrop(sender, DragDropEffects.Move) ==
        DragDropEffects.None)
      (sender as Label).Visible = false;
}
```

Result. If dragging of the label-"animal" ends outside the form (in this case the cursor looks like a prohibition sign), the animal "runs away from the zoo" and its label on the form disappears.

Comment

The DoDragDrop method (which activates the drag-and-drop mode) finishes only when the drag-and-drop mode ends; this method returns the effect that ended the drag. In particular, if it returns the DragDropEffects.None value, it means that the source is dropped over an *invalid target*, that is, at the moment when the mouse cursor looks like a prohibition sign. In our program, all con-

trols on the form (and the form itself) are valid targets, so the DoDragDrop method will return DragDropEffects.None only when the source is dropped *outside the form*.

10.5. Additional coloring of source and target while dragging

Modify the label1_MouseDown and label1_DragEnter methods as follows:

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
  {
    Label src = sender as Label;
    src.ForeColor = Color.Blue;
    if (DoDragDrop(sender, DragDropEffects.Move) ==
        DragDropEffects.None)
      (sender as Label).Visible = false;
      src.Visible = false;
    src.ForeColor = SystemColors.ControlText;
  }
}
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
  (sender as Control).BackColor = Color.Yellow;
}
```

Add the following statement to the label1_DragDrop and textBox1_DragDrop methods:

```
trg.BackColor = SystemColors.Control;
```

Define an event handler for the DragLeave event for label1 and then connect the created handler to the DragLeave events of label2 – label4 and textBox1 – textBox4.

label1.DragLeave handler

```
private void label1_DragLeave(object sender, EventArgs e)
{
  (sender as Control).BackColor = SystemColors.Control;
}
```

Result. In the drag-and-drop mode, the text color of the source label changes to blue, the current target control (label or text box) is displayed on a yellow background (see Comment 1).

Disadvantage. When you *click* on any of the labels, its background color changes to yellow.

Correction. In the label1_DragDrop method, move the statement you just added to it

```
trg.BackColor = SystemColors.Control;
```

to a position in front of the statement

```
if (src == trg) return;
```

Result. Now clicking on the label does not change its appearance (see Comment 2).

Comments

1. For additional highlighting of a drag-and-drop source, it is enough to adjust its properties *before* calling the DoDragDrop method. The changed properties should be restored *after* finishing this method (that is, after ending the drag-and-drop mode). To highlight the current target, change its properties in the DragEnter event handler and restore it in the DragLeave event handler, which occurs when the mouse cursor leaves the current target. Note that in the case of an invalid target, the DragLeave event occures when the drag-and-drop mode ends over this target. If the drag-and-drop mode finishes over a valid target, then the DragLeave event does not occur, and the DragDrop event handler must be used to restore the target properties.

Note that the label1_DragEnter and label1_DragLeave handlers are used for both labels and text boxes. This is possible because the Control class (which is the common ancestor of both labels and text boxes) has a BackColor property that we want to change. Thus, in these handlers, it suffices to cast the sender parameter to the Control type. In the case of the label1_DragLeave handler, we also take advantage of the fact that labels and text boxes (in read-only mode) have the *same* standard SystemColors.Control background color.

2. The noted disadvantage is related to the fact that, when the button is clicked, the MouseDown event occurs and the label1_MouseDown handler for this event *starts* the drag-and-drop mode, which ends immediately when the mouse button is released. In this case, the label becames a drag-and-drop target, that is, the DragEnter event occures for it, leading to the background color changing. However, the background color is not restored in the label1_DragDrop handler, since the exit from this handler occurs immediately after the src == trg condition is satisfied. Thus, to correct the disadvantage, it is enough to restore the background color *before* checking this condition.

10.6. Customizing the cursor in drag-and-drop mode

Define the GiveFeedback event handler for Form1:

Form1.GiveFeedback handler

```
private void Form1_GiveFeedback(object sender,
  GiveFeedbackEventArgs e)
{
  e.UseDefaultCursors = false;
  Cursor.Current = e.Effect == DragDropEffects.Move ?
    Cursors.Hand : Cursors.No;
```

```
}
```

Result. In the drag-and-drop mode, the mouse cursor looks like a "pointing hand" or, outside the form, like a prohibition sign.

Comments

1. The GiveFeedback event of the Form class is provided to change the appearance of the cursor in the drag-and-drop mode. You can determine the current drag effect in this event handler using the e.Effect property. Depending on the effect, you can set the new cursor, but before that you must disable the display of standard drag-and-drop cursors by setting the e.UseDefaultCursors property to false. The required cursor must be assigned to the Current static property of the Cursor class.

2. The .NET help system says that the Cursor.Current static property allows to determine and change the current cursor. This is only partly true. Indeed, calling this property allows to find out how the cursor looks at a given time in the application. However, you can change the appearance of the cursor using the Cursor.Current property only in the MouseMove event handler. If you assign a new value to the Cursor.Current property elsewhere in the program, then this value *will be replaced with the default value* as soon as the MouseMove event occurs in the program. For this reason, this property is rarely used, although the Cursor.Current property should be changed in the GiveFeedBack event handler.

10.7. Information about the current state of the program. Buttons with images

Place the button1 on Form1, specify **The zoo is closed** as its title (that is, the Text property) and adjust the position of the button in accordance with Fig. 10.3.

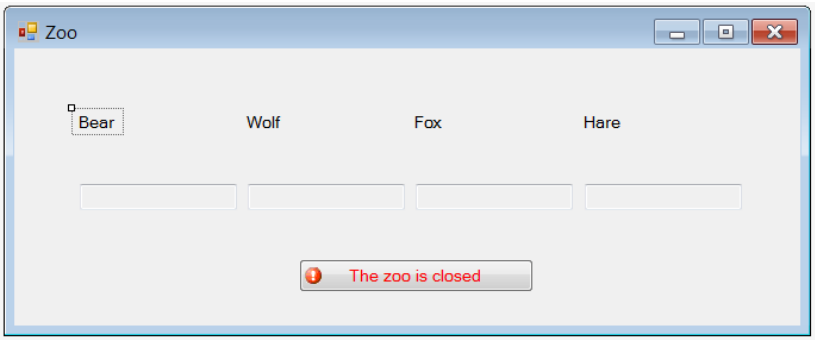


Fig. 10.3. The final view of Form1

For greater clarity, add a small image to the left of the button title. This image (like the button title) will depend on the current state of the program.

You can download the **Visual Studio Image Library** from Microsoft site (web link https://www.microsoft.com/en-us/download/details.aspx?id=35825). This collection contains images for buttons associated with various actions. We will use the **Visual Studio 2012 Image Library**. After downloading the **VS2012 Image Library.zip** file with the collection archive, you just need to unzip it.

A feature of the **Visual Studio 2012 Image Library** collection is that it also contains a set of pictures for the version 2010 of Visual Studio (this set is contained in the **x - archive - x** subdirectory). The pictures in the 2010 set are less abstract and more descriptive.

In our project, we will use the **Error.bmp**  and **GoLtrHS.bmp**  files located in the **x - archive - x\Objects - VS2012\bmp_format\Office and VS** subdirectory. To simplify actions to connect these files to the project, it is advisable to copy the files to the project directory.

Add an ImageList control (named imageList1) to the form. This control is intended to hold a set of uniformly sized images. To add the **Error.bmp** and **GoLtrHS.bmp** images to the imageList1 control, follow these steps:

- select the Images property in the **Properties** window of the imageList1 control and click the button  near this property;
- in the **Images Collection Editor** window that appears on screen, click the **Add** button and select the **Error.bmp** file in the **Open** window that appears (as a result, the **Error.bmp** file will be added to the Images collection);
- add the **GoLtrHS.bmp** image to the collection using the same actions and then close the **Images Collection Editor** window by clicking the **OK** button or pressing the Enter key.

Configure the rest of the required properties of the imageList1 and button1 controls (when setting the ImageAlign property of the button1 control, select the square located in the middle- left corner in the panel that appears):

Properties

```
imageList1: TransparentColor = Magenta
button1: ForeColor = Red, ImageList = imageList1,
  ImageKey = Error.bmp, ImageAlign = MiddleLeft
```

Modify the label1_MouseDown method by adding the following statements:

```
string s = "";
for (int i = 1; i <= 4; i++)
{
  if (Controls["label" + i].Visible)
  // at least one of the animal is free
    return;
  s = s + (Controls["textBox" + i].Text);
}
if (s == "")
  // all cages are empty
  return;
button1.Text = "The zoo is open";
button1.ForeColor = Color.Green;
```

```
button1.ImageKey = "GoLtrHS.bmp";
```

Result. If all labels on the form have disappeared and at least one text box contains text (the name of some animal), then the button displays the text **the Zoo is open** and the color of the button changes from red to green (see Comments 1 and 2).

Remark. The new version of the label1_MouseDown method uses the Controls collection property of the form to iterate over *all* labels and *all* text boxes.

Disadvantage. Dragging a label onto button1 causes the label to become invisible since it is overlapped by the button.

Correction. Set the AllowDrop property of button1 to **True** and connect the Form1_DragEnter handler to the button's DragEnter event.

Result. Now, when you try to drag a label onto button1, nothing happens: the label remains in the same place (see Comment 3).

Comments

1. Pay attention to the TransparentColor property of the ImageList control. In a bmp-image intended to be placed on button, one of the colors (usually very bright and therefore not using) plays the role of an *indicator of a transparent color*: this color marks those fragments of the image through which the control containing this image should be visible. In the Visual Studio 2010 Image Library, the color *magenta* (R = 255, G = 0, B = 255) is used as this indicator of a transparent color (in the KnownColor enumeration, this color is represented by two synonym elements: Magenta and Fuchsia). If the TransparentColor property is left at the default value (Transparent), then excess magenta fragments will be displayed on the edges of the images.

2. To indicate which image from the ImageList set should be connected to the control, you can use not only the ImageKey string property containing the name of the image (as in the last statement in the modified label1_MouseDown method), but also the ImageIndex integer property containing the *index* of the image in the set (elements are indexed from 0). So, you could connect the **GoLtrHS.bmp** image to button1 using the button1.ImageImdex = 1 statement. Indices are convenient to use when connecting different images to several controls in a loop.

3. The noted disadvantage is due to the fact that by default the AllowDrop property of the button is equal to **False** and therefore the button is *invisible* for the drag-and-drop mode. Note that simply changing the value of the AllowDrop property to **True** *is not enough*: in this case, the button will behave like an invalid target, and therefore the label released over the button will *disappear* from the form, since this is the action that our program performs when we try to release the source over an invalid target (see Section 10.4). Connecting the button's DragEnter event to the Form1_DragEnter handler solves the problem, since, in this case, the button becomes a valid target with the **Move** effect, alt-

hough *it does nothing* when the source is released over it (since there is no DragDrop event handler associated with the button).

10.8. Restoring the initial state

Define handlers for the Load event for Form1 and the Click event for button1:

Form1.Load and button1.Click handlers

```
private void Form1_Load(object sender, EventArgs e)
{
  for (int i = 1; i <= 4; i++)
    Controls["label" + i].Location =
      Controls["textBox" + i].Location -
      new Size(0, textBox1.Top / 2);
  ActiveControl = button1;
}

private void button1_Click(object sender, EventArgs e)
{
  Form1_Load(this, null);
  for (int i = 1; i <= 4; i++)
  {
    Controls["label" + i].Visible = true;
    Control c = Controls["textBox" + i];
    c.Text = "";
    c.Tag = 0;
  }
  button1.Text = "The zoo is closed";
  button1.ForeColor = Color.Red;
  button1.ImageKey = "Error.bmp";
}
```

Result. The initial position of the labels-"animals" is now determined programmatically, namely, in the Load event handler of the form (the labels are located above the corresponding text boxes and aligned to their left border). Further, when you press button1, the initial position of the "animals" is restored and the "cages" are released. In addition, when the program starts, the button1 becomes active, which results in a red border around it (the color of the border around the active button is determined by the ForeColor color of its caption).

Remark. In these methods, the items from the Controls collection are not cast to their actual type (Label or TextBox). This is not necessary, since all the control properties used in these methods are already defined in the Control class, which is the *common ancestor* of all visual controls (recall that the Controls collection returns elements of Control type).

Comment

The Load event of a form occurs only once, after the form's constructor finishes but before the form is displayed on the screen. This event is especially useful if situations are possible when, after starting the program, instead of displaying the form, you need to display an error message and immediately terminate the program (here is an example of such a situation: the demo program found that the trial time has expired; see also Section 23.5). Although such situations can be detected already in the form's constructor, you cannot call the Close method from the form's constructor since this will result in a run-time error. In order to correctly terminate the program before the form appears on the screen, you should call the Close method in the Load event handler.

It also makes sense to place a piece of code that performs some initializing actions in the Load event handler (not in the form's constructor) if the program may need to *repeat these initializing actions in the future*. In such a situation, it is sufficient to explicitly call the Load event handler, as is done in the button1_Click method.

11. Cursors and icons: CURSORS project

The CURSORS project introduces the use of cursors (the Cursor and Cursors classes) and icons (the Icon class) in an application. It also provides an introduction to generics and how to embed graphical resources into an application. Also we consider the NotifyIcon control for working with the traybar of the Windows taskbar.

11.1. Using standard cursors

After creating the CURSORS project, place the button (named button1) on Form1 and configure the properties of the form and the added button (see also Fig. 11.1):

Properties

```
Form1: Text = Cursors and Icons, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
button1: Text = Default
```

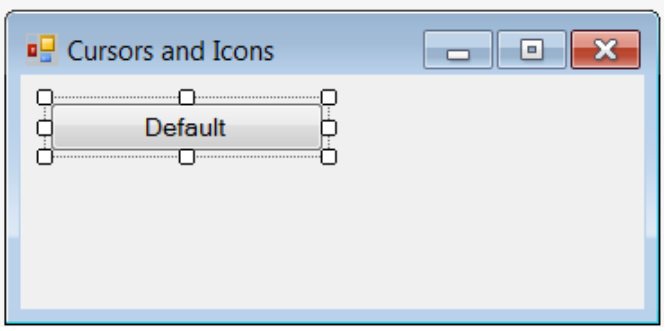


Fig. 11.1. Form1 view at the initial stage of development

Add two field declarations to the beginning of the Form1 class declaration:

```
private List<string> str = new List<string>(30);
private List<Cursor> cur = new List<Cursor>(30);
```

Add new statements to the constructor of the Form1 class:

```
public Form1()
{
  InitializeComponent();
  foreach (System.Reflection.PropertyInfo pi in
    typeof(Cursors).GetProperties())
  {
    str.Add(pi.Name);
    cur.Add((Cursor)pi.GetValue(null, null));
  }
```

```
  button1.Tag = str.IndexOf("Default");
}
```

Define an event handler for the MouseDown event for button1:

button1.MouseDown handler

```
private void button1_MouseDown(object sender, MouseEventArgs e)
{
  int k = (int)button1.Tag,
    c = str.Count;
  switch (e.Button)
  {
    case MouseButtons.Left:
      k = (k + 1) % c; break;
    case MouseButtons.Right:
      k = (k - 1 + c) % c; break;
  }
  button1.Text = str[k];
  button1.Cursor = cur[k];
  button1.Tag = k;
}
```

Result. Clicking on the **Default** button changes the cursor for this button; this action also changes the button title to the name of the current cursor. All 28 standard cursors are selected cyclically; the order of cursors corresponds to the order of their position in the drop-down list for the Cursor property in the **Properties** window. When you press not the left mouse button, but the right mouse button, the cursors are selected in reverse order.

Comments

1. All standard cursors can be accessed using the Cursors class, which has a special read-only property for each standard cursor. The Cursors class lacks the facility to loop through the standard cursors or to retrieve their names. Therefore, in the constructor of the form, we create two *collections* of the *generic* List<T> type: the str collection, which contains the *names* of all the standard cursors, and the cur collection, which contains the *cursors* themselves (that is, objects of Cursor type). Collections are formed using the *reflection mechanism* applied to the Cursors class. The reflection mechanism is not considered in detail in this book (you can familiarize yourself with it, for example, in [3, Chapter 19]); we only note that it defines an array of PropertyInfo objects, each of which contains information about a property of the Cursors class, and then the foreach loop iterates over these objects and extracts information about the names and values of the corresponding properties of the Cursors class.

2. With the generic classes introduced in .NET 2.0, you can easily create *typed collections* (such as the List<T> class) defined in the System.Collections.

Generic namespace. When declaring collections of List<T> type, you must immediately specify the type T of their elements. In our program, the str collection is declared as List<string>, that is, it is intended for storing string elements, and the cur collection is declared as List<Cursor> and is intended for storing elements of Cursor type.

In earlier versions of .NET, you could only use collections that were considered to be objects of the base object type (an example of such a collection is the ArrayList class defined in the System.Collections namespace). The use of such collections was potentially dangerous, since errors related to mismatch of type of elements added to the collection could be detected only at the stage of application execution. In addition, using such collections was inconvenient, since, when accessing their elements, it was required to *explicitly convert* the elements to the required type. *Generic collections* are more reliable (they cannot contain data whose type differs from the type of collection elements, since errors of this kind are detected already at the compilation stage) and are more convenient to use (since when accessing collection elements, there is no need to perform type conversion). Chapter 7 of the book [3] is devoted to a detailed examination of generics; see also Chapter 9 in [2].

3. We save the index of the cursor associated with button1 in the Tag property of this button. Any visual control has the Tag property; it can be used to store some of its additional characteristics (see also Section 10.3). Since the Tag property is of object type, it can be assigned a value of any type; however, when reading its value, you must perform an explicit conversion to the actual type of the object it contains (in our case, to the int type). The last statement in the form's constructor sets the Tag property of button1 to the index of the element in the cur collection that corresponds to the default cursor.

4. In the button1_MouseDown method, looping through the indices in the range from 0 to c (where c = str.Count - 1 and str.Count is equal to the number of elements contained in the str collection) is performed using the % operator (that is, *taking modulo* operator, or *taking remainder* operator). An additional term c in the expression (k - 1 + c)% c has been added so that the expression in parentheses *never takes negative values*.

5. To handle clicks on button1, the MouseDown event is used, since the Click event of the Button control reacts only to the *left* mouse button (for other controls, the situation may be different; for example, the Click event of the Label control reacts to a click of any mouse button – see Section 7.3).

11.2. Setting the cursor for a form and waiting mode indication

Place three new buttons (button2, button3, button4) on Form1 and set the Text properties of these buttons to the **Form Cursor**, **Wait Cursor**, and **Default Form Cursor**, respectively (Fig. 11.2).

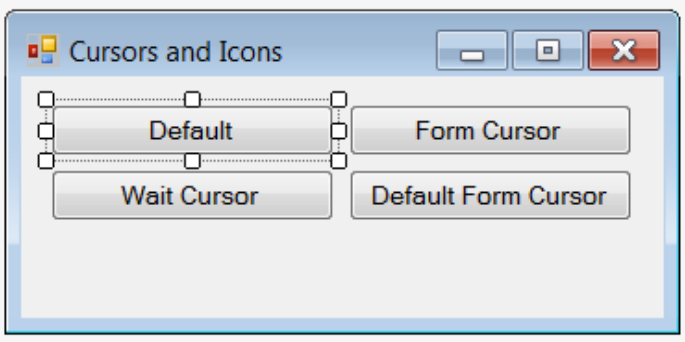


Fig. 11.2. Form1 view at an intermediate stage of development

Define event handlers for the Click event for the added buttons:

button2.Click, button3.Click, and button4.Click handlers

```
private void button2_Click(object sender, EventArgs e)
{
  Cursor = button1.Cursor;
}
private void button3_Click(object sender, EventArgs e)
{
  UseWaitCursor = true;
}
private void button4_Click(object sender, EventArgs e)
{
  UseWaitCursor = false;
  Cursor = Cursors.Default;
}
```

Result. Clicking the **Form Cursor** button propagates the action of the new cursor to the entire form including the buttons on it. Clicking the **Wait Cursor** button sets the *wait cursor* (usually looking as hourglass) for the entire form and its controls. Clicking the **Default Form Cursor** button cancels the wait cursor and restores the default cursor for the form (but not for the first button whose cursor still matches the name given in its title).

Comments

1. If the Cursor property of the visual control is null or has the Cursors.Default value, then the cursor type for this control is determined by its *parent* control (in our case, the form); for a form, the type of cursor with the Default value is determined by the *operating system*. That is why, when changing the cursor for a form, the cursor automatically changes for button2, button3, and button4 buttons, whose Cursor property is equal to Default.

2. Boolean property UseWaitCursor is available for any visual control. If the UseWaitCursor property of any control is set equal to true, then the current cursor of this control *and all its child controls* will be replaced with a wait cursor (hourglass).

11.3. Connecting new cursors to the project and saving them as embedded resources

In addition to standard cursors, you can use other cursors in the program. Cursor files have the .**cur** extension; they can be found, for example, in the **Cursors** subdirectory of the Windows directory. It should be noted that animated cursors (files with the .**ani** extension) cannot be used in the Windows Forms applications, and colored cursors are displayed as monochrome when used in such applications. The icon of any cur-file corresponds to the image of the cursor contained in that file; this allows you to quickly familiarize yourself with the contents of cur-files using the **Explorer** or **My Computer** applications.

Select any two cur-files containing monochrome cursors and copy them into the CURSORS project directory under the names **C1.cur** and **C2.cur**.

Execute the **Project | Add Existing Item…** menu command. In the dialog box that appears, enter the name of the **C1.cur** file (you can do this directly in the text box; you can also specify the **All Files** option in the drop-down list of file types and select the **C1.cur** file from the list of all files contained in the directory). Press the Enter key or the **Add** button. This will add the **C1.cur** file to the CURSORS project, as you can verify by looking at the **Solution Explorer** window.

Select the **C1.cur** file in the **Solution Explorer** window; this will display its properties in the **Properties** window. Set the Build Action property to **Embedded Resource**; no other properties need to be changed.

Add the **C2.cur** file to the project in the same way and set its Build Action property.

Add new statements to the constructor of the Form1 class:

```
for (int i = 1; i <= 2; i++)
{
  str.Add("C" + i);
  cur.Add(new Cursor(GetType(), "C" + i + ".cur"));
}
```

Result. When creating a form, new cursors are loaded into the program and added to the list of available cursors under the names **C1** and **C2**. Further they are processed in the same way as standard cursors (see Sections 11.1 and 11.2). It should be emphasized that the images of these cursors are located directly in the **CURSORS.exe** executable file.

Comments

1. To load the cursor from the application resources, a version of the Cursor class constructor with two parameters is used; the first parameter usually has the form GetType(), and the second specifies the name of the corresponding resource. You can also load the cursor directly from the cur-file; the corresponding constructor of the Cursor class has one parameter, which is the file name (if

the cur-file is contained in the working directory of the application, then the path to the file can be omitted).

2. The Visual Studio environment has built-in tools to create new cursors and edit existing ones. To create a new cursor (and immediately bind it to the current project), just execute the **Project | Add New Item…** menu command. In the window that appears, specify the type of the new element (**Cursor file**) and its name, then press the Enter key or the **Add** button. A template for the monochrome cursor with the specified name will be created and loaded immediately into the environment editor.

You can edit the cursor image using various tools whose shortkey buttons are displayed on the **Image Editor** toolbar. To select the color of lines and background, the **Colors** window is used; the button for displaying the **Colors** window is located at the left border of the Visual Studio window. In the **Colors** window, the color of the lines is selected with the left mouse button, the background is selected with the right mouse button. In addition to black-and-white patterns that simulate shades of gray, you can select two special colors: green (corresponds to the *transparent* part of the cursor) and pink (corresponds to the part of the cursor, under which the colors will be *inverted*).

For the created cursor, you need to specify the *hot spot*, that is, the pixel that determines the position of the cursor on the screen (for example, in the case of a standard arrow cursor, its hot spot is the arrowhead). The hot spot is defined using the **Set Hot Spot Tool** button  on the **Image Editor** toolbar. In addition, the coordinates of the hot spot are specified (and can be changed) in the **Properties** window.

To edit an existing cursor connected to the project, just double-click on its name in the **Solution Explorer** window; as a result, the selected cursor will be loaded into the image editor.

11.4. Working with icons

Small images called *icons* are stored in files with the .**ico** extension. Many ico-files are contained in the **Visual Studio Image Library** (see Section 10.7); also it is easy to find ico-files on your computer and on the internet. The icon for an ico-file, like the icon for a cur-file, corresponds to the image it contains.

Adding existing ico-files to the project is similar to adding cur-files. The following steps assume that the **Computer.ico** and **Folder.ico** files have been added to the project (in the **Visual Studio 2012 Image Library**, they are located, for instance, in the **x--archive--x\Objects - VS2012\ico_format\WinVista** subdirectory). As with cur-files, after adding the ico-file to the project, set its Biuld Action property to **Embedded Resource**.

Place a new button (named button5) on the form and set its Text property to **Icon 0**. Add the declaration of the ico array to the beginning of the Form1 class declaration:

```
private Icon[] ico = new Icon[3];
```

Add new statements to the constructor of Form1:

```
button5.Tag = 0;
ico[1] = new Icon(GetType(), "Computer.ico");
ico[2] = new Icon(GetType(), "Folder.ico");
ico[0] = Icon;
```

Define an event handler for the Click event for button5:

button5.Click handler

```
private void button5_Click(object sender, EventArgs e)
{
  int k = ((int)button5.Tag + 1) % 3;
  button5.Text = "Icon " + k;
  button5.Tag = k;
  Icon = ico[k];
}
```

Result. The new **Icon 0** button changes the icon of the form both in its title and on its button located on the Windows taskbar. Icon numbered 0 corresponds to the initial *default icon* of the form. The program uses a fixed-size ico array to store three icons.

Comments

1. The new fragments of the program code show that working with icons requires the same techniques as working with cursors; you just need to use the Icon class and the Icon property of the form. You can also download the icon directly from the ico-file; for this, the Icon class provides a constructor with one parameter specifying the file name.

2. The icon can be associated not only with the form, but also with the application itself; however, the application icon is used only when the corresponding exe-file is displayed in **Explorer** or similar file browsers that support the display of icons. To set the application icon, open the tab with project properties in the Visual Studio editor using the **Project | *<project name>* Properties…** menu command and select the required icon from the **Icon** drop-down list located in the **Application** section.

11.5. Placing an icon of application in the notification area

Place a new button (named button6) on the form, as well as a NotifyIcon control (this control will be named notifyIcon1), and set their properties (see also Fig. 11.3):

Properties

```
button6: Text = Show Tray Icon
notifyIcon1: Text = Icon in Traybar, Visible = False
```

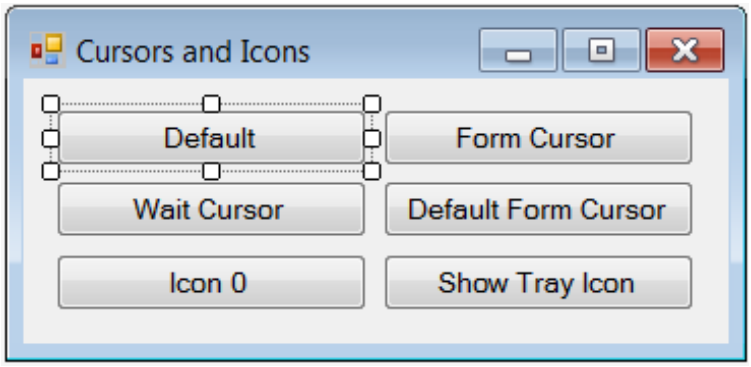


Fig. 11.3. The final view of Form1

Note that the NotifyIcon control is not a visual control, so, when you add it to a form, it is placed in a special area of non-visual controls located below the form.

In the constructor of the Form1 class, modify the last statement as follows:

```
notifyIcon1.Icon = ico[0] = Icon;
```

In the button5_Click method, modify the last statement as well:

```
notifyIcon1.Icon = Icon = ico[k];
```

Define the Click event handler for button6:

```
private void button6_Click(object sender, EventArgs e)
{
  bool b = button6.Text == "Show Tray Icon";
  notifyIcon1.Visible = b;
  ShowInTaskbar = !b;
  button6.Text = b ? "Hide Tray Icon" : "Show Tray Icon";
}
```

Result. Clicking button6 hides the form button on the Windows taskbar and displays its icon in the notification area (*traybar*) on the right side of the Windows taskbar. When you move the mouse cursor over this icon, the **Icon in Traybar** tooltip appears. Pressing button6 again restores the initial form button on the Windows taskbar.

Comments

1. Statements changed in the constructor of the Form1 class and in the button5_Click method are of the form variable1 = variable2 = expression. Statements of this kind allow you to simultaneously assign the specified expression to all the variables located before it. This possibility is due to the fact that an assignment operator of the form v = expr not only assigns the value of the expression expr to the variable v, but also *returns the assigned value*. In addition, it should be taken into account that, if there are several assignment operators, they are performed from right to left, that is, in the notation a = b = c, parentheses are implicitly placed as follows: a = (b = c).

2. Typically, the notification area contains icons for applications running in the background mode and using windows only to show and change their set-

tings. In addition to tooltips that appear when the cursor is moved over the icon, the NotifyIcon control allows you to display a *message in the balloon* near the icon (using the ShowBalloonTip method and related properties BalloonTipIcon, BalloonTipTitle, BalloonTipText). The NotifyIcon control can also display a *context menu* using the right mouse button (the ContextMenuStrip property; working with context menus is described in Section 14.3). The notification area icon can also respond to mouse events; to this purpose, the NotifyIcon control provides a number of events including Click and DblClick.

12. Menus and processing of text files: TEXTEDIT1 project

The TEXTEDIT1 project is the first in a series of projects related to the development of a fully functional text editor. This project describes the creation and configuration of the main application menu (the MenuStrip control) and implements the basic actions with text files (creating, saving, loading, as well as displaying a request to save changes). The features of the dialog controls SaveFileDialog and OpenFileDialog and the multi-line version of the TextBox control are considered.

12.1. Menu creation

After creating the TEXTEDIT1 project, place the MenuStrip and TextBox controls on Form1 (the added controls will be named menuStrip1 and textBox1) and set properties of the form and the added controls:

Properties

```
Form1: Text = Text Editor, StartPosition = CenterScreen
textBox1: Text = empty string, Multiline = True, Dock = Fill
```

Note that the MenuStrip and TextBox controls should be placed on the form *in the specified order*; otherwise (if you first place the TextBox control on the form), in some early versions of the Windows Forms library, the top of the text box may be hidden under the menu bar. However, such a mistake can be easily fixed: it is enough to send the MenuStrip control to the back using the **Send to Back** button of the **Layout** panel (this panel can be displayed on the screen using the menu command **View | Toolbars | Layout**).

To create a menu in design mode, the *menu designer* is used, which is activated when the menuStrip1 control is selected. In the text box with the **Type Here** text, input the name of the created menu item. After pressing the Enter key, a new item will be created in the menu. In addition, new text boxes with the **Type Here** text will appear next to and below the created item, allowing you to create new menu items of the first or second level (see Fig. 12.1; this figure shows the menu items that will be added later). When a created menu item is selected, the **Properties** window displays the properties of this item.

Each MenuStrip item is a ToolStripMenuItem object. The name of the menu item (unlike the names of other controls placed on the form) is obtained not by adding an order number to the type name (for example, label1), but by adding the *name of the menu item* to the *left* of the type name, for example, fileToolStripMenuItem. As a result, very long names are obtained, so immediately after creating a menu item, it is advisable to change the name of the item by

specifying the new value of its Name property. Note that the Name property is located in the **Properties** window at the top of the property list and its title is enclosed in parentheses: **(Name)**.

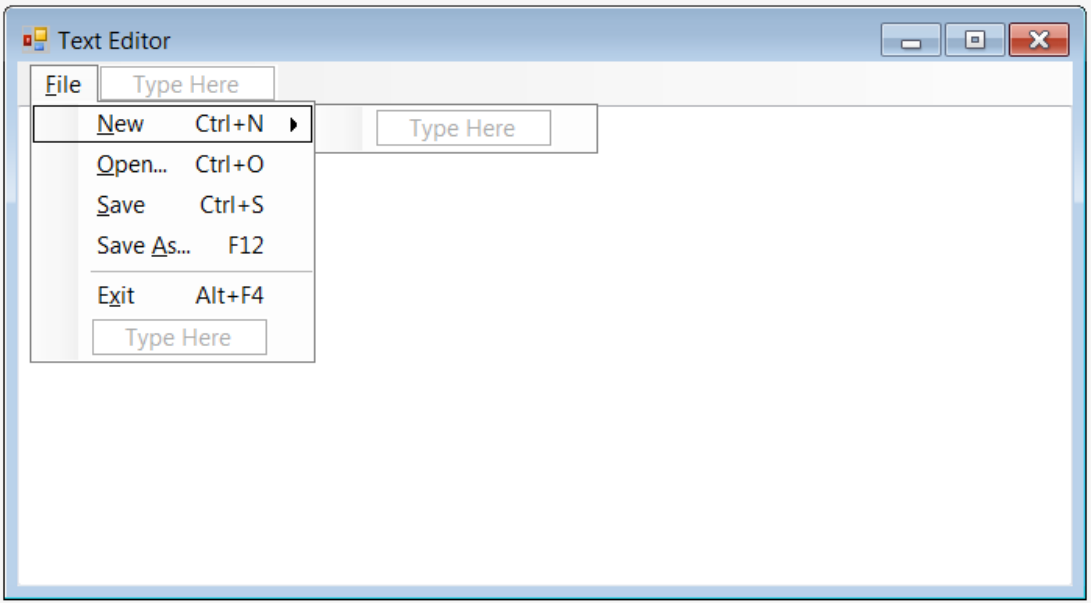


Figure: 12.1. Form1 view with the menu designer

In order to make the relation between the menu item and its name clearer, we will use, as the menu item name, the text of this item typed with a small letter and supplemented with the number 1 (for example, for the **File** item, we specify the file1 name). The number allows us to avoid possible conflicts with the key words of C#; in addition, we will further associate some commands not only with menu items but also with shortcut buttons or context menu items; for the names of these controls, it will be convenient to use the name of the same command supplemented with a different number (see Sections 14.3 and 15.1).

Create a first-level **&File** menu item in the menuStrip1 control and use the **Properties** window to change the name of this item (that is, the Name property) to file1. In the drop-down menu associated with the **File** menu item, create four items with the following text: **&New**, **&Open**, **&Save**, **Save &As**. Then create an item with the text - (*dash*); this item will be converted to a horizontal *separator*. Below the separator, create another menu item with the **E&xit** text. Set the properties of the added menu items as follows (note that the values for the ShortcutKeys property are easier and faster to input directly from the keyboard than using the drop-down list):

```
New (the File group): Name = new1, ShortcutKeys = Ctrl+N
Open (the File group): Name = open1, ShortcutKeys = Ctrl+O
Save (the File group): Name = save1, ShortcutKeys = Ctrl+S
Save As (the File group): Name = saveAs1, ShortcutKeys = F12
Exit (the File group): Name = exit1, ShortcutKeys = Alt+F4
```

The final view of the **File** group menu is shown in Fig. 12.1.

Using the menu designer, you can also define handlers for events associated with the selected menu item. Select the exit1 menu item and define the Click event handler for it:

exit1.Click handler

```
private void exit1_Click(object sender, EventArgs e)
{
  Close();
}
```

To exit the menu designer, it is enough to select the form itself or some of its other controls.

Result. Due to the **Fill** value of the Dock property of the textBox1 control, the edit area is automatically resized when the window is resized. The program has a menu with items that can be called using the mouse, shortcut keys (Ctrl+N, F12, etc.), as well as with using the key combinations Alt+*underlined letter in the item text* (if the second-level menu is expanded, then, to select an item, just press the key corresponding to the underlined character in the item text without pressing the Alt key). The **Exit** command closes the application, the other menu commands do not perform any action yet.

Remark. When starting the program, the text of the first-level menu items may not contain underscores. In this case, press the Alt key and, without releasing it, wait for the underscore characters to appear; then press the key corresponding to the underlined character. Also note that you can simply press the F10 or Alt key to go to the menu.

Comments

1. Exit from the program by pressing the Alt+F4 key combination is performed automatically, so there is no need to explicitly specify the shortcut key for the **Exit** command. However, such an option increases the clarity of the program, since it provides the display of the shortcut key name next to the corresponding menu item.

2. Although the menuStrip1 control is a descendant of the Control class and therefore is a visual control, it is located not on the form in design mode, but in the area *below* it (recall that this area is intended for placement of *non-visual controls*). At the same time, the menu that the menuStrip1 control defines is displayed on the form. You can select the MenuStrip control by clicking either on its image in the area of non-visual controls or on the image of the menu in the form (outside the existing menu items, since, when you click on a menu item, this item is selected instead of the menuStrip1 control itself). When the MenuStrip control is selected, its properties are displayed in the **Properties** window.

3. The MenuStrip control appeared in version 2.0 of the .NET Framework library. Unlike its predecessor, the MainMenu control, which was an object wrapper for the Windows traditional menu, the MenuStrip control is not associ-

ated with the Windows menu and is a "usual" visual control. This allows, in particular, to dock the MenuStrip control to any window border and provides more flexible customization of the menu's appearance (for example, you can change its font by simply setting the Font property, which is not possible for the MainMenu control). Items of the MenuStrip menu (which are objects of ToolStripMenuItem type) have a richer set of properties than the items of the "old" MainMenu control. In particular, they have the Tag property, as well as the Image property, which provides adding an *icon* to the text of a menu item. For these reasons, the old MainMenu control has been removed from the list of controls in the **Toolbox** window; moreover, this control is not available in .NET Core 3.1 and later versions.

12.2. Saving text to a file

Add a non-visual control of SaveFileDialog type (named saveFileDialog1) to Form1; this control will be placed in the non-visual control area below the form image. Set the properties of the added control (see Comment 1 for the DefaultExt and Filter properties):

Properties

```
saveFileDialog1: DefaultExt = txt, Title = Save file
  Filter = Text files|*.txt
```

At the beginning of the **Form1.cs** file, insert the directive for introducing the namespace for classes related to the file input-output:

```
using System.IO;
```

In the constructor of the Form1 class, add a statement:

```
saveFileDialog1.InitialDirectory =
  Directory.GetCurrentDirectory();
```

Add a new SaveToFile method to the Form1 class:

```
private void SaveToFile(string path)
{
  File.WriteAllText(path, textBox1.Text);
}
```

Define the Click event handlers for the saveAs1 and save1 menu items:

saveAs1.Click and save1.Click handlers

```
private void saveAs1_Click(object sender, EventArgs e)
{
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string path = saveFileDialog1.FileName;
    SaveToFile(path);
    Text = "Text Editor - " + Path.GetFileName(path);
  }
```

```
}
private void save1_Click(object sender, EventArgs e)
{
  string path = saveFileDialog1.FileName;
  if (path == "")
    saveAs1_Click(saveAs1, null);
  else
    SaveToFile(path);
}
```

Result. When the **Save As...** command is executed, the **Save file** dialog box appears (the program's working directory is selected as the initial directory for file saving). If you specify a file name in the dialog box and click the **Save** button or press the Enter key, then the text from the editor will be saved in this file, and the file name will appear in the title bar of the program window. By default, the file name is supplied with the .**txt** extension. When executing the **Save** command, the file name is not prompted, unless the text has not yet been saved.

When you exit the **Save file** dialog box by clicking the **Cancel** button or pressing the Esc key, the text is not saved to the file. When you try to save the text under the name of an existing file, you are prompted to confirm this action. If you specify a directory that does not exist, a warning message is displayed, but the dialog box does not close and the error can be corrected immediately.

Comments

1. To display on the screen a *standard dialog box* associated with the SaveFileDialog control (or the OpenFileDialog control which will be considered later), you must call the ShowDialog method of this control. If the dialog box is closed by the **Save** button (for SaveFileDialog) or by the **Open** button (for OpenFileDialog), then the ShowDialog method returns DialogResult.OK and the name of the selected file is saved in the FileName property. If the dialog is closed by the **Cancel** button, then the ShowDialog method returns DialogResult.Cancel. Note that *all* forms have the ShowDialog method (see Sections 5.1 and 5.4).

Of the frequently used properties of the SaveFileDialog and OpenFileDialog controls, note the following:

- DefaultExt (the default file extension; the dot at the beginning of the extension is not indicated);
- Filter (filters for files displayed in the dialog box; for each filter, its description is first indicated and then, after the vertical bar |, the filter itself indicated as a list of file masks separated by semicolons; the symbol | also separates filters from each other);
- InitialDirectory (the initial directory displayed when the dialog box is opened).

2. The name of the current file is saved in the saveFileDialog1.FileName property. The GetFileName method of the Path class extracts the name and extension from the full file name (the drive and path are discarded).

3. To write data to a text file, we use the WriteAllText method of the File class from the System.IO namespace. When storing text, it takes into account all end-of-line markers. The first parameter specifies the file name, the second parameter specifies the saved text. The WriteAllText method can have a third, optional parameter that specifies the *encoding* in which the text is stored in the file. If this parameter is not specified, then UTF-8 encoding is used.

4. To automatically handle errors related to an attempt to overwrite an existing file and an attempt to specify a nonexistent directory, the OverwritePrompt and CheckPathExists properties of the saveFileDialog1 control must be set to **True** (this is their default value).

5. Despite the error control provided by the dialog box, when saving a file (as well as opening it, which will be implemented in the next section), many various error situations may occur. For their handling, it is strongly recommended that you use the try–catch statement (see Chapter 3). In this and subsequent projects, we do not use the try–catch statement just because we do not want to increase the size of the program code.

Disadvantage. When the file is subsequently saved by the **Save As** command, the *full path* to this file is displayed in the dialog box, which complicates its editing. Note that in standard programs, dialog boxes never display full filenames.

Correction. Add the following statement to the beginning of the saveAs1_Click method:

```
saveFileDialog1.FileName =
  Path.GetFileName(saveFileDialog1.FileName);
```

Result. Now the dialog box displays only the name and extension of the previously saved file.

Error. After the correction was made, the program does not work correctly if the **Save file** dialog was called and the user closed it without selecting the file. In this situation, the short file name (without the file path) is saved in the saveFileDialog1.FileName property, and therefore, when the **Save** command is subsequently executed (in which the dialog box is not displayed), the file is saved not in its original directory, but in the *current* directory of application.

Correction. Modify the saveAs1_Click method as follows:

```
private void saveAs1_Click(object sender, EventArgs e)
{
  string oldPath = saveFileDialog1.FileName;
  saveFileDialog1.FileName =
    Path.GetFileName(saveFileDialog1.FileName);
  saveFileDialog1.FileName = Path.GetFileName(oldPath);
```

```
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string path = saveFileDialog1.FileName;
    SaveToFile(path);
    Text = "Text Editor - " + Path.GetFileName(path);
  }
  else
    saveFileDialog1.FileName = oldPath;
}
```

Result. Now, the above error does not occur because when exiting the saveAs1_Click method, the saveFileDialog1.FileName property *always* contains the full name of the current file.

12.3. Clearing the editing area and opening an existing file

Add a non-visual control of OpenFileDialog type (named openFileDialog1) to Form1; this control will be placed in the non-visual control area below the form image). Set the properties of the added control:

Properties

```
openFileDialog1: DefaultExt = txt, FileName = empty string,
  Title = Open file, Filter = Text files|*.txt
```

Modify the last statement in the constructor of the Form1 class as follows:

```
openFileDialog1.InitialDirectory =
  saveFileDialog1.InitialDirectory =
  Directory.GetCurrentDirectory();
```

Define event handlers for the Click event for the new1 and open1 menu items:

new1.Click and open1.Click handlers

```
private void new1_Click(object sender, EventArgs e)
{
  textBox1.Clear();
  Text = "Text Editor";
  saveFileDialog1.FileName = "";
}
private void open1_Click(object sender, EventArgs e)
{
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string path = openFileDialog1.FileName;
    textBox1.Text = File.ReadAllText(path);
    Text = "Text Editor - " + Path.GetFileName(path);
```

```
    saveFileDialog1.FileName = path;
    openFileDialog1.FileName = "";
  }
}
```

Result. When the **New** command is executed, the editing area is cleared; when the **Open...** command is executed, the **Open file** dialog box appears allowing you to select a file to load into the editor. The working directory of the application is set as the starting directory for the **Open file** dialog box. When you try to open a non-existent file, you receive a warning message, but the dialog box does not close and you can correct the error immediately.

Remark. If the length of a text line from the loaded file excess the width of the editing area, this line is automatically wrapped to the next screen line (the wrapping is performed after one of the space characters).

Comments

1. In the new1_Click method, the saveFileDialog1.FileName property is cleared. This indicates that the new document has not yet been saved to the file. Clearing the openFileDialog1.FileName property in the open1_Click method prevents the name of an already loaded file from being displayed in the **File open** dialog box the next time this dialog box is called.

2. To read data from a text file, the ReadAllText method of the File class is used, which reads the entire contents of the specified file into a string. The resulting string includes end-of-line markers, which allows the read text to be correctly split into separate lines when displayed in the textBox1 control.

When reading data, as well as writing it (see the previous section), our program uses the default UTF-8 encoding.

3. For automatic handling of an error related to an attempt to read a non-existent file, the CheckFileExists property of the openFileDialog1 control must be **True** (this is its default value).

4. Setting the Title property for the SaveFileDialog and OpenFileDialog controls is optional. If this property is not set (that is, it is empty), then the titles of the dialog boxes are defined by the Windows operating system.

5. The WordWrap property of the TextBox control is responsible for the automatic wrap of long lines. Wrapping is performed if this property is **True** (the default value). If you turn off automatic wrapping by setting the WordWrap property to **False**, the trailing portion of long lines may not appear in the editing area. In this situation, a *horizontal scroll bar* may be useful; this scroll bar can be added to a TextBox by setting its ScrollBars property to **Horizontal**. For any value of the WordWrap property, it is also useful to add a *vertical scroll bar* to the TextBox control (to add only a vertical bar, the ScrollBars property should be set to **Vertical**; to add both scroll bars, it should be set to **Both**). The only inconvenience of the vertical bar for the TextBox control is that it is *always* dis-

played on the screen (even if the loaded text does not exceed the visible editing area).

12.4. Request to save changes

Add the following statement to the SaveToFile method of the Form1 class:

```
textBox1.Modified = false;
```

Add a new TextSaved method to the Form1 class:

```
private bool TextSaved()
{
  if (textBox1.Modified)
    switch (MessageBox.Show("Save changes in the document?",
      "Confirmation", MessageBoxButtons.YesNoCancel,
      MessageBoxIcon.Question))
    {
      case DialogResult.Yes:
        save1_Click(this, null);
        return !textBox1.Modified;
      case DialogResult.Cancel:
        return false;
    }
  return true;
}
```

Modify the new1_Click method as follows:

```
private void new1_Click(object sender, EventArgs e)
{
  if (TextSaved())
  {
    textBox1.Clear();
    Text = "Text Editor";
    saveFileDialog1.FileName = "";
  }
}
```

Add a new statement at the beginning of the open1_Click method:

```
if (TextSaved())
```

In addition, define an event handler for the FormClosing event for Form1:

Form1.FormClosing handler

```
private void Form1_FormClosing(object sender,
  FormClosingEventArgs e)
{
  e.Cancel = !TextSaved();
}
```

Result. If changes have been made to the current text, an attempt to clear the editor window with the **New** command, to open a new file with the **Open** command, or to exit the program leads to a prompt asking whether to save the changes. When you click the **Yes** button, the current text is saved under the previous name (if it has never been saved, its name is requested in the **Save file** dialog box). Changes are not saved when you click the **No** button. When the **Cancel** button is clicked, the selected action (**New**, **Open**, or exiting the program) is canceled and the user can continue editing the current text.

Comments

1. The TextSaved method uses the Modified property of the textBox1 control, which is set to true if the text has been modified *by the user*. Note that any programmatic changes to the Text property of the textBox1 control set its Modified property to false. Saving text to a file does not change the Modified property, so a statement has been added to the SaveToFile method to set the Modified property to false.

2. The TextSaved function returns true if the current text was saved to disk or the user explicitly refused to save by clicking the **No** button. If the **Cancel** button is clicked, then the function returns false (this means that the user wants to go back to editing the current text). Pay attention to the statement

```
return !textBox1.Modified;
```

This statement ensures the correct handling of the following situation: the user has not previously saved the text in the file; when prompted to save the text, the user selected **Yes**, but *exited* the appeared **Save file** dialog box by clicking **Cancel**. In this case, the TextSaved function will return false, which is correct because it corresponds the **Cancel** option that the user finally chose.

13. Advanced menu options, color and font setting: TEXTEDIT2 project

The TEXTEDIT2 project continues a series of projects related to the development of a fully functional text editor. This project describes how to create advanced menu commands (checkboxes and radio buttons) and menu groups of the third level. The implementation of commands for setting various font characteristics and text alignment in the editor is considered; we discuss the corresponding properties of the Font and TextBox classes. Also we consider the ColorDialog and FontDialog non-visual controls that create standard dialog boxes for color and font setting.

13.1. Setting the font style (menu items as checkboxes)

The previously developed TEXTEDIT1 project (see Chapter 12) should be used as a template for the TEXTEDIT2 project. Copy the TEXTEDIT1 project to the new TEXTEDIT2 directory and follow the steps required to rename the project (see Section 1.1).

By following steps similar to those used to create a **File** menu item and its associated second-level menu items (see Section 12.1), create a new first-level menu item with the text **F&ormat** in the menuStrip1 control and use the **Properties** window to change the name of this item (that is, its Name property) to format1. In the second-level menu associated with **Format**, create three items with text **&Bold**, **&Italic**, **&Underline** and name bold1, italic1, underline1, respectively. Set the properties of the added menu items (take attention to the Font property, for which you need to change one of its subproperties: Bold, Italic, or Underline):

Properties

```
Bold (the Format group): Name = bold1, ShortcutKeys = Ctrl+B,
  Font.Bold = True
Italic (the Format group): Name = italic1, ShortcutKeys = Ctrl+I,
  Font.Italic = True
Underline (the Format group): Name = underline1,
  ShortcutKeys = Ctrl+U, Font.Underline = True
```

The resulting menu is shown in Fig. 13.1.

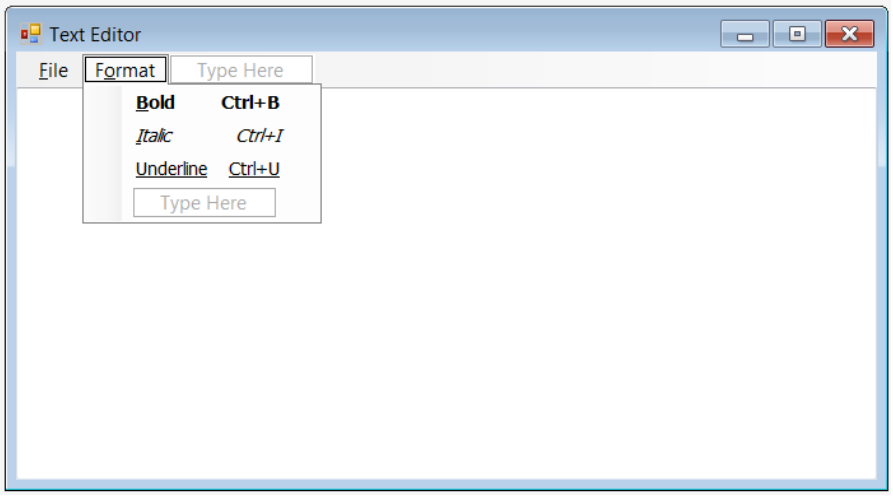


Fig. 13.1. Form1 view at the initial stage of development

Define the Click event handler for the bold1 menu item and then connect the created handler to the italic1 and underline1 menu item Click events (connecting existing handlers to menu item events is the same as connecting to events of usual controls – see Section 6.1):

bold1.Click handler

```
private void bold1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  mi.Checked = !mi.Checked;
  FontStyle fs = textBox1.Font.Style;
  fs = mi.Checked ? (fs | mi.Font.Style) : (fs & ~mi.Font.Style);
  Font f = textBox1.Font;
  textBox1.Font = new Font(f, fs);
  f.Dispose();
}
```

Result. When calling the menu items added to the menu, the corresponding font style of characters in the editor is set as follows: **bold** for the **Bold** command, *italic* for the **Italic** command, underlined for the **Underline** command. When the **Format** menu is called, checkmarks are displayed near the menu items with set styles as for the checkboxes. Indeed, each menu item related with a font style works as a *checkbox*: its first call sets the required font style and its second call cancels this font style. Note that the names of these menu items are displayed in the menu in the font style that they configure (for example, the name of the **Bold** command is displayed in bold style).

Comments

1. The menu items have a CheckOnClick property that allows, when a menu item is called, to automatically switch its state from enabled to disabled, and vice versa. For such a behavior, this property must be set to **True**. By default,

the CheckOnClick property is **False**; in this case, when the menu item is called, its state does not change and additional actions must be taken to change it. For example, in the bold1_Click method, the statement mi.Checked = !mi.Checked performs the required actions. We use explicit changing the state of the menu item because it will simplify the binding of formatting commands to shortcut buttons (see Section 15.2).

2. In order to provide the execution of all three commands using one handler, we used the fact that the font style of the menu item text corresponds to the style that must be added to or removed from the set of font styles of the textBox1 control. To add a required style (that is, an element of the FontStyle enumeration) to a set of styles, we use the operator | (bitwise OR), to remove the style, we use the operator & (bitwise AND) and the operator ~ (bitwise NOT).

3. An additional problem is that the properties of an existing font *cannot be changed*. Therefore, to change the font style, we have to create a *new* font with the specified style. In order for all other font properties to remain unchanged, it is convenient to use the Font class constructor specifically designed to change the font style. This constructor has two parameters (f, s): its first parameter is the existing font f, the second parameter is the required style s. The generated font has all the properties of the font f, with the exception of the style, which is taken from the second parameter of the constructor. Recall that the old font, after creating a new one, should be destroyed by calling its Dispose method.

13.2. Setting text alignment (menu items as radio buttons)

Add new items to the second-level menu associated with the **Format** item (see Section 13.1). Firstly add an item with the text - (dash); this item will be converted to a horizontal separator. Then add three menu items with the text **&Left justify**, **C&enter**, and **&Right justify** and set the properties for those items as follows:

Properties

```
Left justify (the Format group): Name = leftJustify1,
  ShortcutKeys = Ctrl+L, CheckState = Indeterminate
Center (the Format group): Name = center1, ShortcutKeys = Ctrl+E
Right justify (the Format group): Name = rightJustify1,
  ShortcutKeys = Ctrl+R
```

At the beginning of the Form1 class declaration, add a declaration of the new field:

```
private ToolStripMenuItem alignItem;
```

Add new statements to the constructor of the Form1 class:

```
alignItem = leftJustify1;
leftJustify1.Tag = HorizontalAlignment.Left;
center1.Tag = HorizontalAlignment.Center;
```

```
rightJustify1.Tag = HorizontalAlignment.Right;
```

Define the Click event handler for the leftJustify1 menu item and then connect the created handler to the Click event of the center1 and rightJustify1 menu items.

leftJustify1.Click handler

```
private void leftJustify1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  if (mi.Checked) return;
  alignItem.Checked = false;
  alignItem = mi;
  mi.CheckState = CheckState.Indeterminate;
  textBox1.TextAlign = (HorizontalAlignment)mi.Tag;
}
```

Result. When calling the menu items added to the menu, the text alignment in the editor is set as follows: left alignment for **Left justify**, centering for **Center**, right alignment for **Right justify**. When the **Format** menu is called, a mark • is displayed near the command with the current alignment (as for the selected radio button). Thus, the added menu items behave like a *group of radio buttons*.

Comments

1. To set a mark • near a menu item, you should set its CheckState property to **Indeterminate**, but this action will not reset the mark near the previously marked menu item. The alignItem field, which was added to the Form1 class, stores the *currently marked menu item*. If another alignment option is selected, the mark is removed from the previously marked item, the new item is marked, and this item is stored in the alignItem field (see the leftJustify1_Click method). Note that the Checked and CheckState properties are related as follows: Checked is **False** when CheckState has the **Unchecked** value; for other CheckState values, the Checked property is **True**.

2. In order to provide the execution of all three commands using one handler, we used the Tag property. For each menu item, this property contains the alignment value (as some member of the HorizontalAlignment enumeration) that corresponds to that item. Since only string values can be assigned to the Tag property using the **Properties** window, the required assignments are performed in the form's constructor. In the leftJustify1_Click event handler, the Tag property of the executed command is assigned to the TextAlign property of the textBox1 control. Note that the Tag properties of the leftJustify1, rightJustify1, and center1 menu items could be initialized by the numbers 0, 1, and 2, since the corresponding elements of the HorizontalAlignment enumeration have such numeric values, but in this case the resulting code would be less descriptive:

```
leftJustify1.Tag = 0;
```

```
    center1.Tag = 2;
    rightJustify1.Tag = 1;
```

13.3. Setting the color of symbols and background color (third-level menu commands and the Color dialog box)

Add a non-visual control of ColorDialog type (named colorDialog1) to Form1 (this control will be placed in the non-visual control area below the form image).

Add new items to the second-level menu associated with the **Format** item (see Sections 13.1 and 13.2): one more splitter and a menu item with the **&Colors** text. Then go to the third-level menu template associated with the **Colors** menu item and add two menu items with the text **&Font color...** and **&Background color...** to it. Set the Name property of the added menu items:

Properties

```
Colors (the Format group): Name = colors1
Font color… (the Colors group): Name = fontColor1
Background color… (the Colors group): Name = backgroundColor1
```

Define the Click event handlers for the fontColor1 and backgroundColor1 menu items:

fontColor1.Click and backgroundColor1.Click handlers

```
private void fontColor1_Click(object sender, EventArgs e)
{
  colorDialog1.Color = textBox1.ForeColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    textBox1.ForeColor = colorDialog1.Color;
}
private void backgroundColor1_Click(object sender, EventArgs e)
{
  colorDialog1.Color = textBox1.BackColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    textBox1.BackColor = colorDialog1.Color;
}
```

The resulting menu of the **Format** group is shown in Fig. 13.2 (the last item of this menu named **Font...** will be added in the next section).

Result. When the command from the **Colors** menu group is executed, the **Color** dialog box is called, which allows you to set the color of symbols (the **Font color...** command) or the background color (the **Background color...** command). When the dialog box is displayed, the current color is selected (outlined by frame). To set a new color, select it and click **OK**.

Remark. If some dialog box appears on the screen during execution of a menu command, it is recommended to indicate an ellipsis (…) at the end of the name of such a command (earlier in our program, ellipsis was used in the names of the **Open...** and **Save As...** commands). The presence of an ellipsis in the

command name means, in particular, that this command can be canceled after its call if the associated dialog box is closed using the **Cancel** button or the Esc key.

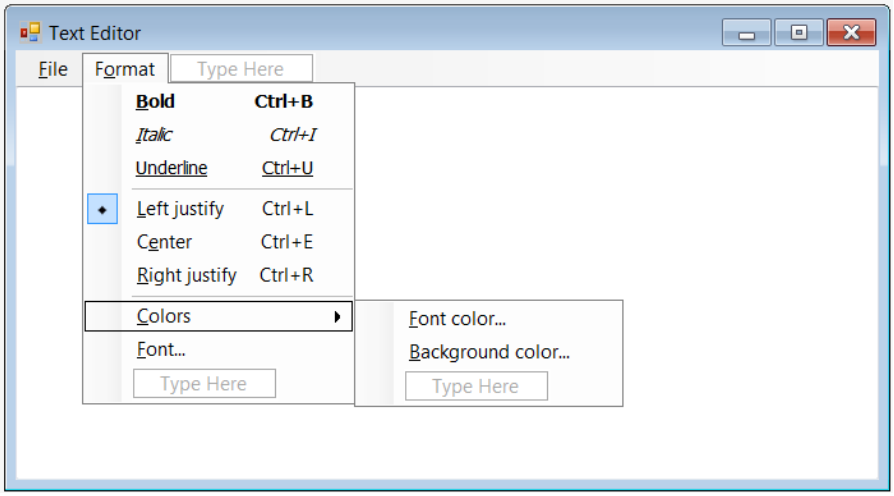


Figure: 13.2. The final view of Form1

13.4. Setting font properties using the Font dialog box

Add a non-visual control of FontDialog type (named fontDialog1) to Form1 (this control will be placed in the area of non-visual controls below the form image).

Add the menu item with the text **&Font…** to the second-level menu associated with the **Format** item (see Fig. 13.2). Set the Name property of the added menu item to font1 and define the Click event handler for that menu item:

font1.Click handler

```
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
  {
    Font f = textBox1.Font;
    textBox1.Font = fontDialog1.Font;
    f.Dispose();
    bold1.Checked = fontDialog1.Font.Bold;
    italic1.Checked = fontDialog1.Font.Italic;
    underline1.Checked = fontDialog1.Font.Underline;
  }
}
```

Result. When the **Font...** command is executed, the **Font** dialog box is called, allowing you to change the font in the editor. When the dialog box is displayed, it indicates the current font settings. If you change the font style, the

checkboxes for the **Bold**, **Italic**, and **Underline** menu items are additionally adjusted. See also Comment 1.

Error. If you open the **Font** dialog box and, without changing any font properties, close the window by clicking **OK** or pressing the Enter key, then a runtime error will occur *the next time* you open the **Font** window. This error occurs "inside" the ShowDialog method and is related to the interaction of the Font properties of the fontDialog1 and textBox1 objects, in particular, with disposing the old font instance for the textBox1 control. You can come to such a conclusion by noticing that if you comment out the f.Dispose() statement, then the specified error does not occur.

So, to correct the error, you may *remove* call of the Dispose method. But this is a bad solution, as it does not free up system resources used by an old font instance. Of course, in the absence of other alternatives, this solution may be used, but we can choose the best way.

Correction. Add one more conditional statement to the font1_Click method as follows:

```
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
    if (!textBox1.Font.Equals(fontDialog1.Font))
    {
      Font f = textBox1.Font;
      textBox1.Font = fontDialog1.Font;
      f.Dispose();
      bold1.Checked = fontDialog1.Font.Bold;
      italic1.Checked = fontDialog1.Font.Italic;
      underline1.Checked = fontDialog1.Font.Underline;
    }
}
```

Result. Now, in the situation described above, no runtime error occurs (see Comment 2).

Remark. It should be admitted that the used way of setting the font in some (extremely rare) situations will still lead to a runtime error. The fact is that among the standard Windows fonts there are fonts that do not implement all font styles. For example, the **Monotype Corsiva** font has no regular (non-italic) style defined. You cannot set invalid styles for such fonts using the **Font** window, but you can try to do this later using the font style settings from the **Format** menu. Trying to do this will result in a runtime error. Since there are very few such fonts available, we will not check this special situation in our project. The reader can implement such processing himself by enclosing all the bold1_Click method statements, except for the first one, in a try block, and by placing the error mes-

sage **The specified style combination is not available for the current font** in the catch section of this try block, as well as the statement for restoring the previous state of the selected menu item: mi.Checked = !mi.Checked. Note also that using the FontFamily class and its IsStyleAvailable method, you can determine which styles are available for a given font.

Comments

1. In order to determine in the font1_Click method whether the required style is set for the selected font, we used the following Boolean properties of the Font class: Bold, Italic, Underline. Recall that these properties (like all other properties of the Font class) are read-only.

2. The conditional statement added in the new version of the font1_Click method allows to bypass all actions to change the font in the case when the initial font was not changed in the dialog box. The Equals method allows to check that two objects are *equal*, that is, that they describe the same fonts. Note that the == operator cannot be used in this situation, since for many classes (including the Font class) the == operator returns true only if both objects being compared are *identical*, that is, they are references to the *same object* allocated in memory. If you try to use the == operator instead of the Equals method in our case, you will always obtain the false result; in other words, after the ShowDialog method is executed, the Font properties of the fontDialog1 and textBox1 objects will always refer to *different* objects. Thus, these properties will never be identical, but they can be equal.

14. Editing commands, context menus: TEXTEDIT3 project

The TEXTEDIT3 project continues a series of projects related to the development of a fully functional text editor. This project implements standard text editing commands that are included in both the main menu (the MenuStrip control) and the editor's context menu (the ContextMenuStrip control). Also we discuss issues related to the use of the Windows clipboard in .NET applications (the Clipboard class).

14.1. Editing commands

The previously developed TEXTEDIT2 project (see Chapter 13) should be used as a template for the TEXTEDIT3 project.

Go to the menu designer (see Section 12.1) and insert a new first-level menu item *before* the already existing **Format** item. To do this, select the **Format** item, call its context menu (by right-clicking), and select the **Insert | MenuItem** command. The new menu item will have a name toolStripMenuItem*N* (where *N* is some number) and a title corresponding to this name. Change the title of the new menu item to **&Edit** (this can be done in the menu designer itself) and replace its name, that is, the Name property, with edit1 using the **Properties** window.

In the second-level menu associated with the new **Edit** item, create menu items with the following text: **&Undo**, a dash - (this item will be converted to a separator), **Cu&t**, **&Copy**, **&Paste**, **&Delete**, another dash -, **&Select All**. Set the properties of the added menu items; as a result, the **Edit** menu will take the form shown in Fig. 14.1.

Properties

```
Undo (the Edit group): Name = undo1, ShortcutKeys = Ctrl+Z
Cut (the Edit group): Name = cut1, ShortcutKeys = Ctrl+X
Copy (the Edit group): Name = copy1, ShortcutKeys = Ctrl+C
Paste (the Edit group): Name = paste1, ShortcutKeys = Ctrl+V
Delete (the Edit group): Name = delete1
Select All (the Edit group): Name = selectAll1,
  ShortcutKeys = Ctrl+A
```

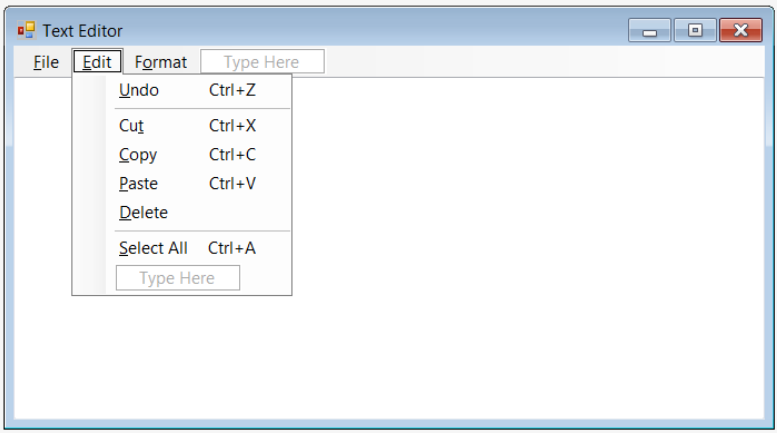


Fig. 14.1. View of Form1 with the expanded **Edit** menu

Define click event handlers for the **Edit** menu items:

undo1.Click, cut1.Click, copy1.Click, paste1.Click, delete1.Click, selectAll1.Click handlers

```
private void undo1_Click(object sender, EventArgs e)
{
  textBox1.Undo();
}
private void cut1_Click(object sender, EventArgs e)
{
  textBox1.Cut();
}
private void copy1_Click(object sender, EventArgs e)
{
  textBox1.Copy();
}
private void paste1_Click(object sender, EventArgs e)
{
  textBox1.Paste();
}
private void delete1_Click(object sender, EventArgs e)
{
  textBox1.SelectedText = "";
}
private void selectAll1_Click(object sender, EventArgs e)
{
  textBox1.SelectAll();
}
```

Result. The menu contains standard *editing commands*: the **Undo** command to undo the last editor action, the **Cut** and **Copy** command to cut and copy the selected fragment to the clipboard, the **Paste** command to paste a fragment

from the clipboard, the **Delete** command to delete the selected fragment, the **Select All** command to select all the text in the editor.

Comments

1. The Delete key is the default hot key for the deleting a selected text (that is, it is handled in any TextBox control). In the case when the text does not contain a selection, a certain action is also provided for the Delete key (deleting the character to the right of the caret), so you should not specify it in the delete1.ShortcutKeys property. The hot keys Ctrl+Z, Ctrl+X, Ctrl+C, Ctrl+V are also handled as required in any TextBox control. By specifying them in the corresponding ShortcutKeys properties, we only ensured that these hot keys are displayed in the menu. All of the above commands are also available from the *standard context menu* of the TextBox control (the context menu is invoked by right-clicking on the control).

2. It should be noted that those of the implemented editing commands that change the content of the textBox1 control always set its Modified property to true, whereas programmatical changing the Text property set the Modified property to false.

14.2. Special visualization of unavailable editing commands. Working with the clipboard

Define the Click event handler for the edit1 menu item:

edit1.Click handler

```
private void edit1_Click(object sender, EventArgs e)
{
  undo1.Enabled = textBox1.CanUndo;
  cut1.Enabled = copy1.Enabled = delete1.Enabled =
    textBox1.SelectionLength > 0;
  paste1.Enabled =
    Clipboard.GetDataObject().GetDataPresent(typeof(string));
  selectAll1.Enabled = textBox1.Text != "";
}
```

Result. When the submenu of the **Edit** menu group is called, the unavailable menu items are grayed out. The **Undo** item is available if an action was previously performed that can be undone. The **Cut**, **Copy, Delete** items are available if the text contains a selection. The **Paste** item is available if the clipboard contains data in text format (see also the comment). The **Select All** item is available if the editor contains non-empty text.

Disadvantage. The edit1_Click handler is executed *after* expanding the submenu of the **Edit** command, so the user can see how the appearance of inaccessible menu items changes.

Correction. In the **Events** tab of the **Properties** window for the edit1 control, do the following:

- *remove* the text **edit1_Click** from the **Click** text box;
- *add* the text **edit1_Click** to the **DropDownOpening** text box (the easiest way to do this is to use the drop-down list for this text box).

Result. Now the actions specified in the handler are performed after clicking on the **Edit** menu item, but *before* expanding the submenu, since the DropDownOpening event occurs at this moment.

Comment

To check for the presence of a text fragment in the clipboard, we used the GetDataObject static method of the Clipboard class. This method returns an object of IDataObject type. Using methods of this object, we can check for the presence of data of the type of interest in the Windows clipboard (using the GetDataPresent Boolean function) and also get this data (using the GetData function, which returns a value of object type). Both functions have one parameter of Type type, which determines the required data type (in our case, string). To check that the clipboard contains a bitmap or vector image, call the GetDataPresent method with the typeof(Bitmap) or typeof(Metafile) parameter, respectively. Note that the clipboard can store any .NET object (for example, a button), but only a .NET application which places this .NET object on the clipboard can get such an object (at the same time, strings and images that have been placed on the clipboard are available for *all* applications).

The clipboard can simultaneously store data of *different* formats, so the fact that the GetDataPresent(typeof(string)) method returns true does not mean that the GetDataPresent(typeof(Bitmap)) method will necessarily return false.

The SetDataObject static method of the Clipboard class is used to place a data item on the clipboard. As the first parameter of this method, you must specify an object containing the required data item (for example, a string). A *copy* of the item is placed on the clipboard; this action removes old data of the same type from the clipboard. The SetDataObject method can have a second parameter of bool type. If this parameter is true, the data item will be contained in the clipboard even after the termination of the application that placed it on the clipboard. It is clear that this should only be done for data that can be recognized by *all* applications (for example, strings or images). If the second parameter is false or absent, then the data item is automatically removed from the clipboard when the application terminates.

It is also possible to place *several representations* of the same data item on the clipboard (for example, a program that places text in RTF format on the clipboard can simultaneously place a non-formatted text). The application can determine all available formats of data placed on the clipboard and get data of the required format. For these purposes, the GetDataPresent and GetData functions are equipped with overloaded versions that take as a parameter not a type, but a *text string* describing the required format (for example, "Rich Text For-

mat"). All possible formats are represented in the DataFormat class as static read-only string fields.

14.3. Creating a context menu

Add a control of ContextMenuStrip type (named contextMenuStrip1) to Form1 (this control will be placed in the area of non-visual controls below the form image). This control allows using *context menus* in the application. Bind the contextMenuStrip1 control to the edit area (that is, the textBox1 control) by setting the contextMenuStrip property of the textBox1 control to contextMenuStrip1.

To define the items of the context menu, as in the case of a usual menu, the menu designer is used. However, unlike a usual menu, the context menu designer is displayed on the form only when the control associated with the context menu is selected (in our case, this is the contextMenuStrip1 control). You cannot create a horizontal first-level menu for a context menu; it is also not recommended to create nested drop-down menus in the context menu and to associate shortcut keys with the commands of the context menu.

Proceeding in the same way as in Section 12.1 when creating the **File** menu item and its associated second-level menu items, create menu items in the contextMenuStrip1 context menu. The text of these menu items should be as follows: **Cu&t**, **&Copy**, **&Paste**, a dash - (which will be converted to a separator), **&Font...** (see Fig.14.2).

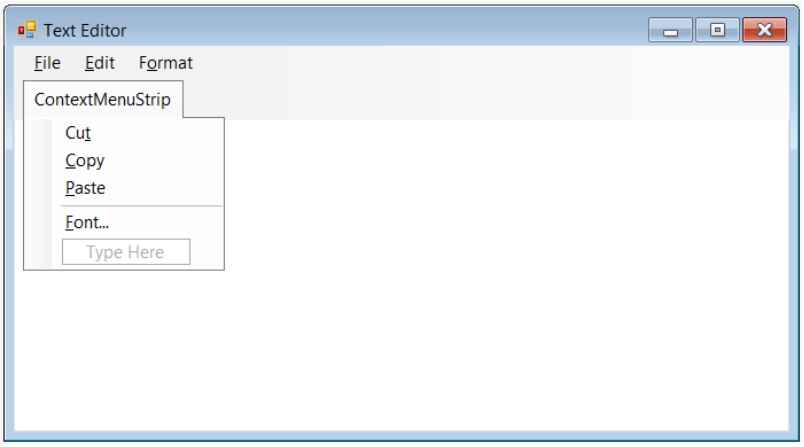


Fig. 14.2. Form1 view with the context menu constructor

Set the properties of the added menu items (note that when defining the names for the context menu items, we do not use the suffix 1, as for the main menu items, but the suffix 2):

Properties

```
Cut (the contextMenuStrip1 menu): Name = cut2,
  Click = cut1_Click
Copy (the contextMenuStrip1 menu): Name = copy2,
  Click = copy1_Click
Paste (the contextMenuStrip1 menu): Name = paste2,
```

```
  Click = paste1_Click
Font (the contextMenuStrip1 menu): Name = font2,
  Click = font1_Click
```

In the list of properties, we also indicated *existing handlers* which should be connected to the Click event for each of the context menu items.

In addition, define an Opening event handler for the contextMenuStrip1 control:

contextMenuStrip1.Opening handler

```
private void contextMenuStrip1_Opening(object sender,
  CancelEventArgs e)
{
  cut2.Enabled = copy2.Enabled = textBox1.SelectionLength > 0;
  paste2.Enabled =
    Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
```

Result. When you right-click in the editor area of the window, the context menu defined in the contextMenuStrip1 control is displayed instead of the standard context menu of the TextBox control. The context menu items that are currently unavailable are grayed out.

15. Toolbar: TEXTEDIT4 project

The TEXTEDIT4 project continues a series of projects related to the development of a fully functional text editor. This project creates and configures the application *toolbar* (the ToolStrip control) and its elements: usual shortcut buttons and shortcut buttons that behave as checkboxes and radio buttons. We describe the steps required to add images to an application resource file and to associate the added images with shortcut buttons and menu items.

15.1. Creation a toolbar and shortcut buttons. Adding images to menu items

The previously developed TEXTEDIT3 project (see Chapter 14) should be used as a template for the TEXTEDIT4 project.

Place a *toolbar* (that is, a control of ToolStrip type) on Form1; this control will be named toolStrip1. The toolbar will automatically dock to the top border of the form and will be positioned below the main menu menuStrip1. However, it will hide the top of the textBox1 control. To place all three controls without overlapping, we need to arrange them in the correct *z*-order (see Comment 4 in Section 9.1). In our case, the easiest way is to execute the **Bring to Front** command for the textBox1 control. This command (as well as the **Send To Back** command) can be executed by a corresponding button on the **Layout** panel and also by a context menu command that can be called for any visual control placed on the form.

When placed on a form, a toolbar contains a drop-down list of all types of controls that can be placed on that toolbar. In our program, only usual *shortcut buttons* (also called *speed buttons*) and separators will be used. When the **Button** option is selected, a button control of ToolStripButton type is placed on the toolbar and is immediately named (in this case, as toolStripButton1). Since the shortcut buttons will be associated with the corresponding items of the main menu, we will replace the default button name with the name of the corresponding menu item and add the digit 0 to it, for example, new10.

Controls placed on the toolbar automatically line up with no spaces between them. To add a standard separator space, select the **Separator** option from the drop-down list of available toolbar controls. This action adds a control of ToolStripSeparator type to the toolbar and immediately assigns it an appropriate name (for example, toolStripSeparator1). Since there is no need to refer to the separators in the future, we will not change their names. Note that the relative position of buttons and separators on the toolbar can be changed by dragging the required control with the mouse to a new location.

Add the following controls to the toolStrip1 toolbar (in order from left to right; the names of the button controls, that is, their Name properties, must be specified in the **Properties** window): the new10 button, the open10 button, the save10 button, a separator, the cut10 button, the copy10 button, the paste10 button.

By default, the Text property of the shortcut button coincides with its name (that is, the Name property), however, it is not displayed on the button, since it is assumed that the button will not contain text, but an *image* (the DisplayStyle property is responsible for the appearance of the button; its default value is **Image**). All buttons added to the toolbar are associated with a standard image. Of course, it needs to be replaced with an image corresponding to the action that each button should perform.

Before assigning images to buttons, add all the necessary images to the resource file of the project being developed. Let us describe the steps for adding images, assuming that the **Visual Studio 2012 Image Library** is available to us (see Section 10.7).

Select the first button added to the panel (new10) and in the **Properties** window go to the Image property. Click the ellipsis button ... near this property, in the **Select Resource** window that appears, select the **Project resource file** radio button and click the **Import...** button. In the **Open** window that appears, go to the **x--archive--x\Actions - VS2010\24bitcolor bitmaps** subdirectory of the **Visual Studio 2012 Image Library** collection, select the **Document.bmp** file and click **Open** or press Enter. As a result, the name of the added resource will appear in the **Select Resource** window: **Document**. Without closing the **Select Resource** windows, add the following files from the same subdirectory to the resource file: **OpenFolder.bmp**, **Save.bmp**, **Cut.bmp**, **Copy.bmp**, **Paste.bmp**.

After adding all the images, select **Document** from their list and click **OK**. The selected image will be associated with the new10 shortcut button. Follow the same steps to add images to all shortcut buttons. In the list of properties below, we specify the name of the image from the resource list, which should be set to the Image property for each button. In addition, this property list shows which *existing* handler should be connected to the Click property of each button. Another property that should be configured for each button is ToolTipText; this property is responsible for displaying a *tooltip* when the mouse is hovering over the button.

Properties

```
new10: Image = Document, ToolTipText = New, Click = new1_Click
open10: Image = OpenFolder, ToolTipText = Open,
  Click = open1_Click
save10: Image = Save, ToolTipText = Save, Click = save1_Click
cut10: Image = Cut, ToolTipText = Cut, Click = cut1_Click
copy10: Image = Copy, ToolTipText = Copy, Click = copy1_Click
```

```
paste10: Image = Paste, ToolTipText = Paste, Click = paste1_Click
```

Make sure that the ImageTransparentColor property is set to **Magenta** for all shortcut buttons (this color is used in all images for the fragments that should be transparent). The resulting toolbar is shown in Fig. 15.1.

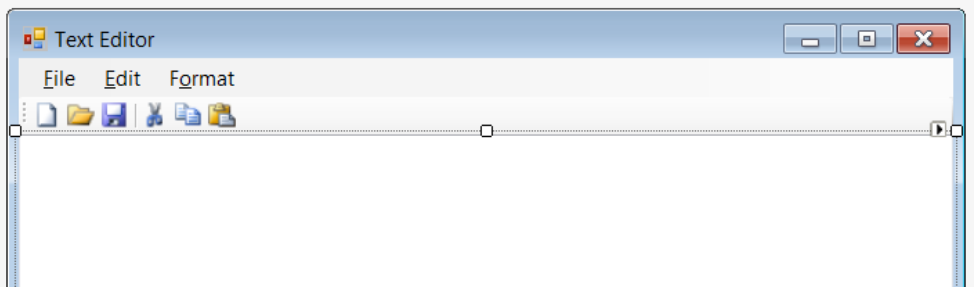


Fig. 15.1. The upper part of Form1 at the first stage of development

Once images are included in the project resource file, they can be used not only for shortcut buttons, but also for main menu items. To do this, simply connect the corresponding image to the Image property of the menu item (in the same way as when connecting an image to a shortcut button). For each menu item, we indicate the name of the image that needs to be associated with it: new1 – **Document**, open1 – **OpenFolder**, save1 – **Save**, cut1 – **Cut**, copy1 – **Copy**, paste1 – **Paste**. All of these menu items must have the ImageTransparentColor property set to **Magenta**.

Result. To execute frequently used commands, it is now enough to click on the corresponding shortcut button. To determine which command the shortcut button is associated with, you need to move the mouse cursor to it: after 1–2 seconds, a yellow tab with the name of the corresponding command will appear next to the button. Menu items that have shortcut buttons display the same images as their associated buttons.

Remark. The presence of images in menu commands allows the user to quickly remember the shortcut button associated with each command.

Comment

You can also use the ImageList control to associate shortcut buttons with images (see Section 10.7). To do this, just place this control on the form (the control will be named imageList1), add the required images to the control, and set the ImageList property of the toolStrip1 toolbar equal to imageList1. For the shortcut buttons, it remains to set the ImageKey properties (similar to how it was done for the usual button in Section 10.7). Unfortunately, the **Properties** window lacks all the properties of the ToolStrip and ToolStripButton controls associated with accessing the ImageList control. Therefore, when using this method of binding images to shortcut buttons, you must *programmatically* set the required properties in the form constructor; in addition, the required images will not appear on the buttons in design mode.

15.2. Using shortcut buttons that behave as checkboxes and radio buttons

In this section, all buttons added to the toolbar will be provided with *text* captions. These captions are specified in the Text property. To display a text caption on a button, the DisplayStyle property of the button must be set equal to Text.

Add new controls to the toolStrip1 toolbar (in order from left to right); the names of the button controls, that is, their Name properties, should be specified in the **Properties** window: separator, bold10 button, italic10 button, underline10 button, separator, leftJustify10 button, center10 button, rightJustify10 button. Set the properties of the added controls. The resulting toolbar is shown in Fig. 21.2.

Properties

```
bold10: Text = B, DisplayStyle = Text, ToolTipText = Bold,
  Font.Bold = True
italic10: Text = I, DisplayStyle = Text, ToolTipText = Italic,
  Font.Italic = True
underline10: Text = U, DisplayStyle = Text,
  ToolTipText = Underline, Font.Underline = True
leftJustify10: Text = <, DisplayStyle = Text,
  ToolTipText = Left justify, Checked = True
center10: Text = =, DisplayStyle = Text, ToolTipText = Center
rightJustify10: Text = >, DisplayStyle = Text,
  ToolTipText = Right justify
```

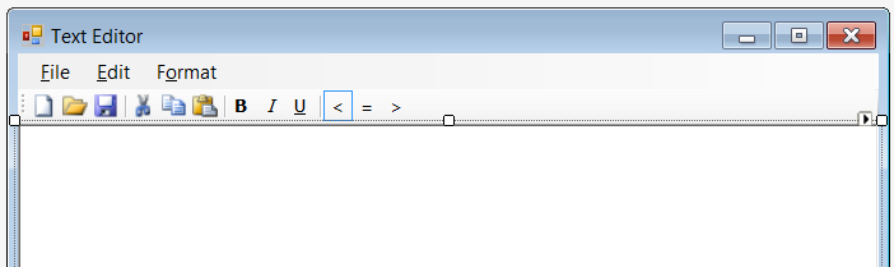


Fig. 15.2. The final version of the upper part of Form1

Remark. Instead of text captions, the follows images from the **Actions\bmp** directory of the **Visual Studio 2012 Image Library** could be connected to the new shortcut buttons: bold10 – **Bold_11689_24.bmp**, italic10 – **Italic_11693_24.bmp**, underline10 – **Underline_11700_24.bmp**, leftJustify10 – **LeftJustify_b11695_24.bmp**, center10 – **Centered_11691_24.bmp**, rightJustify10 – **RightJustify_11699_24.bmp**.

Add the GetButton method to the Form1 class:

```
private ToolStripButton GetButton(ToolStripMenuItem mi)
{
  return toolStrip1.Items[mi.Name + "0"] as ToolStripButton;
}
```

Define handlers for the Click event for the new shortcut buttons: bold10.Click, italic10.Click, underline10.Click, leftJustify10.Click, center10.Click, rightJustify10.Click handlers

```
private void bold10_Click(object sender, EventArgs e)
{
  bold1_Click(bold1, null);
}
private void italic10_Click(object sender, EventArgs e)
{
  bold1_Click(italic1, null);
}
private void underline10_Click(object sender, EventArgs e)
{
  bold1_Click(underline1, null);
}
private void leftJustify10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(leftJustify1, null);
}
private void center10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(center1, null);
}
private void rightJustify10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(rightJustify1, null);
}
```

Modify the bold1_Click, leftJustify1_Click, and font1_Click methods:

```
private void bold1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  mi.Checked = !mi.Checked;
  FontStyle fs = textBox1.Font.Style;
  fs = mi.Checked ? (fs | mi.Font.Style) : (fs & ~mi.Font.Style);
  Font f = textBox1.Font;
  textBox1.Font = new Font(f, fs);
  f.Dispose();
  ToolStripButton sb = GetButton(mi);
  sb.Checked = !sb.Checked;
}
private void leftJustify1_Click(object sender, EventArgs e)
```

```
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  if (mi.Checked) return;
  GetButton(alignItem).Checked = alignItem.Checked = false;
  alignItem = mi;
  mi.CheckState = CheckState.Indeterminate;
  GetButton(mi).Checked = true;
  textBox1.TextAlign = (HorizontalAlignment)mi.Tag;
}
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
    if (!textBox1.Font.Equals(fontDialog1.Font))
    {
      Font f = textBox1.Font;
      textBox1.Font = fontDialog1.Font;
      f.Dispose();
      bold10.Checked = bold1.Checked = fontDialog1.Font.Bold;
      italic10.Checked =
        italic1.Checked = fontDialog1.Font.Italic;
      underline10.Checked =
        underline1.Checked = fontDialog1.Font.Underline;
    }
}
```

Result. To set the font style and text alignment, just click on the corresponding shortcut button, which will become "pressed". In what follows, we will use the expressions "pressed state" and "released state" of a button, although, when using the standard .NET button image style, the "pressed" button simply has an extra border (see the leftJustify10 button image in Fig. 15.2).

The pressed state of the shortcut button means that the specified mode is set. The font setting buttons bold10, italic10, underline10 act as *checkboxes*: each button can be switched to the pressed or released state independently of the others. The text alignment buttons leftJustify10, center10, rightJustify10 act like a *group of radio buttons*: clicking on any of them will release the rest. It should be noted that, when formatting commands are executed directly from the menu or using shortcut keys, the state of the corresponding shortcut buttons is adjusted appropriately.

Comments

1. The GetButton method added to the Form1 class allows to get the associated shortcut button for the menu item. This is possible due to using similar

names for menu items and associated shortcut buttons: to get a shortcut button, it is enough to use the Items collection property of the toolStrip1 control and specify the name of the required button as a key (this name is equal to the name of the corresponding menu command, supplemented with the digit 0). The GetButton method also allows you to check if any shortcut button is associated with a given menu item: if there is no button associated with the item, the method will return null (this additional feature is not used in our program, but it can be useful when implementing interaction of menu items and shortcuts in other applications).

2. In our program, the buttons "delegate" the execution of all actions (including changing their own state) to the menu items. This ensures that the state of the buttons is changed correctly when formatting commands are executed in other ways (from the menu or using shortcut keys). To implement the behavior of the leftJustify10, center10, rightJustify10 controls as a group of radio buttons, the alignItem field is used, with the help of which the selection is removed from the previously selected shortcut button of this group.

16. Status bar and hints: TEXTEDIT5 project

The TEXTEDIT5 project continues a series of projects related to the development of a fully functional text editor. This project creates and configures the status bar of the application (the StatusStrip control) and its label elements. We describe the actions to check the state of the Caps Lock and Num Lock toggle keys, hide and redisplay the toolbar and status bar, and display an expanded hint for the selected menu command on the status bar. In addition, the ToolTip control is described, which allows to associate a contextual hint with any visual control of the application.

16.1. Using the status bar

The previously developed TEXTEDIT4 project (see Chapter 15) should be used as a template for the TEXTEDIT5 project.

Add a *status bar* (that is, a control of StatusStrip type) to Form1; this control will be named statusStrip1. Also add a non-visual control of Timer type, which will be named timer1 and will be placed in the non-visual control area under the form image. To prevent the bottom part of the textBox1 control from being overlapped by the added status bar, you should execute the **Bring to Front** command for the textBox1 control (see Section 15.1).

To add controls to the status bar, a drop-down list is intended, in which we will use only the StatusLabel item (this item adds a *label* control of ToolStripStatusLabel type to the status bar). Since the default names for the controls added to the status bar are too long (for example, toolStripStatusLabel1), we will change them choosing the names according to the purpose of each item in the status bar.

Add the label controls with the specified names to the statusStrip1 status bar (the names of the label controls, that is, their Name properties, should be specified in the **Properties** window): cap1, num1, modified1, hint1. Set the properties of the added label controls, as well as the timer1 control.

Properties

```
cap1: Text = CAP, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 40
num1: Text = NUM, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 40
modified1: Text = Modified, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 80
hint1: Text = Ready, Margin.Left = 5
timer1: Interval = 200, Enabled = True
```

The Size.Width property is used to set the new label width, the Margin.Left property of the hint1 label is used to increase the space between the **Ready** text of this label and the border of the previous modified1 label). The resulting status bar is shown in Fig. 16.1.

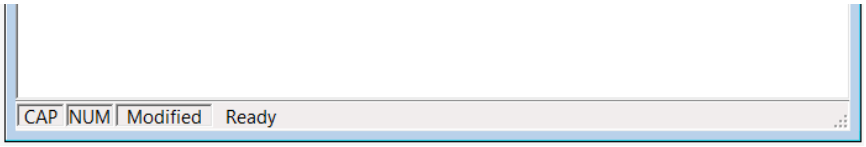


Fig. 16.1. The lower part of Form1

Define an event handler for the Tick event for timer1:

timer1.Tick handler

```
private void timer1_Tick(object sender, EventArgs e)
{
  cap1.Text = IsKeyLocked(Keys.CapsLock) ? "CAP" : "";
  num1.Text = IsKeyLocked(Keys.NumLock) ? "NUM" : "";
  modified1.Text = textBox1.Modified ? "Modified" : "";
}
```

Add a call to that handler to the constructor of the Form1 class:

```
timer1_Tick(this, null);
```

Result. The status bar displays the current state of the Caps Lock and Num Lock keys. In addition, it shows whether the document has been changed since it was last saved (if it has been changed, the string **Modified** is displayed).

Comments

1. To be able to determine the current state of the Caps Lock, Num Lock, and Scroll Lock keys, a new static method IsKeyLocked of boolean type was added to the Control class in version .NET 2.0. Only three members of the Keys enumeration can be specified as a parameter of this method: Keys.CapsLock, Keys.NumLock, and Keys.Scroll; an attempt to specify another member of the Keys enumeration throws a NotSupportedException exception.

2. You can also use the ModifiedChanged event of the textBox1 control to track changes to the Modified property.

3. Calling the Tick event handler in the form constructor provides adjusting the status bar before displaying it on the screen.

16.2. Inaccessible shortcut buttons

Add new statements to the timer1_Tick method:

```
cut10.Enabled = copy10.Enabled = textBox1.SelectionLength > 0;
paste10.Enabled =
  Clipboard.GetDataObject().GetDataPresent(typeof(string));
save1.Enabled = save10.Enabled = textBox1.Modified;
```

Result. Now the state of the shortcut buttons associated with the clipboard matches the state of the corresponding menu items (they are simultaneously

available or unavailable). In addition, the **Save** menu item and its associated shortcut button are now available only after changes have been made to the text being edited.

16.3. Hiding the toolbar and status bar

Proceeding in the same way as in Section 12.1 when creating a **File** menu item and associated second-level menu items, add a first-level menu item with the **&View** text to the menuStrip1 control and use the **Properties** window to change the name of this item (that is, the Name property) to view1. In the second-level menu associated with the **View** item, create items with the text **&Toolbar** and **&Status bar** (Fig. 16.2).

Set the properties of the added menu items:

Properties

```
Toolbar (the View group): Name = toolBar1, Checked = True
Status bar (the View group): Name = statusBar1, Checked = True
```

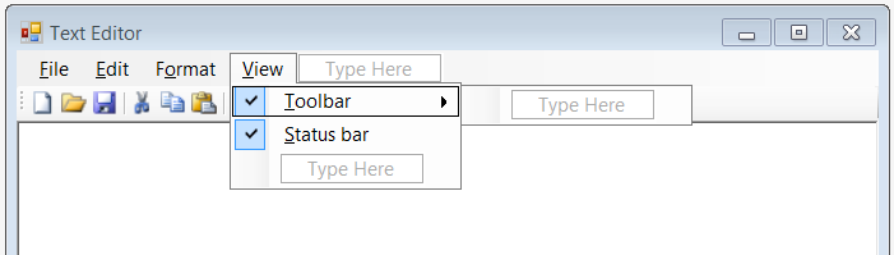


Fig. 16.2. The upper part of Form1

Define Click event handlers for the toolBar1 and statusBar1 menu items:

toolbar1.Click and statusBar1.Click handlers

```
private void toolBar1_Click(object sender, EventArgs e)
{
  toolStrip1.Visible = toolBar1.Checked = !toolBar1.Checked;
}
private void statusBar1_Click(object sender, EventArgs e)
{
  statusStrip1.Visible = statusBar1.Checked =
    !statusBar1.Checked;
}
```

Result. Using the checkbox-style commands of the **View** menu, you can hide and restore the toolbar and status bar.

16.4. Displaying hints on the status bar

Set the ToolTipText property for the menu items in the **File** group:

Properties

```
new1: ToolTipText = Create new document
open1: ToolTipText = Open existing document
```

```
save1: ToolTipText = Save current document
saveAs1: ToolTipText = Save document under new name
exit1: ToolTipText = Exit editor
```

Define event handlers for the MouseEnter and MouseLeave events for the new1 menu item and then connect these handlers to the MouseEnter and MouseLeave events of the remaining menu items in the **File** group. Also connect the new1_MouseLeave handler to the MenuActivate event of the menuStrip1 control.

new1.MouseEnter and new1.MouseLeave handlers

```
private void new1_MouseEnter(object sender, EventArgs e)
{
  hint1.Text = (sender as ToolStripMenuItem).ToolTipText;
}
private void new1_MouseLeave(object sender, EventArgs e)
{
  hint1.Text = "";
}
```

In addition, define an event handler for the MenuDeactivate event for the menuStrip1 control:

menuStrip1.MenuDeactivate handler

```
private void menuStrip1_MenuDeactivate(object sender,
  EventArgs e)
{
  hint1.Text = "Ready";
}
```

Result. When you move the mouse cursor over the **File** menu items, a hint to the selected menu item (if this menu item is available) is displayed on the status bar. For other menu items, the hint is not displayed on the status bar. When exiting the menu, the text **Ready** is restored on the status bar.

Disadvantage. When you move the mouse over a menu item from the **File** group, an additional tooltip appears near this item. The tooltip text duplicates the hint text on the status bar and therefore does not provide any additional information.

Correction. In the constructor of the Form1 class, add a statement that suppresses the display of tooltips for all items of the **File** drop-down menu:

```
file1.DropDown.ShowItemToolTips = false;
```

Remark. You can set the ShowItemToolTips property for the main menu in the **Properties** window for the menuStrip1 control (note that this property is set to **False** by default). However, you can customize it for drop-down submenus (objects of ToolStripDropDown type) only programmatically.

Comments

1. There are certain problems with displaying menu hints in the MenuStrip and ToolStripMenuItem controls. The implementation presented in this section is insufficient, since it does not allow getting hints for menu items by navigating through them using the keyboard. It is interesting to note that the predecessor of the ToolStripMenuItem class, the MenuItem class intended to be used in conjunction with the "old" MainMenu type menu, had the Select event, which was better suited to this task, since it reacted to the selection of a menu item in any way: both with a mouse and a keyboard (however, the event associated with the *loss* of selection was not provided, therefore, to correctly change the tooltips, it was necessary to provide *all* menu items with event handlers for the Select event). Neither the ToolStripMenuItem class nor the MeniStrip class provide events similar to the Select event.

2. Most of the "usual" visual controls (not related to menus, toolbars, and status bars) do not have a property similar to ToolTipText. However, for any visual control, you can show a tooltip by using the ToolTip non-visual control. In addition to a large number of properties that allow you to customize the appearance of the displayed tooltip, the ToolTip control has a SetToolTip(control, hint) method that associates the hint string with the visual control. The GetToolTip(control) method allows you to *get* the value of the tooltip associated with the control visual control. After adding the ToolTip control to the form, all visual controls of the form will display a property of the form **ToolTip on toolTip1** in the **Properties** window. This property allows you to easily adjust tooltip strings in design mode. Using the Popup event of the ToolTip control, you can redirect the display of tooltip text to other controls (for example, to the status bar).

17. Formatting a document: TEXTEDIT6 project

The TEXTEDIT6 project finishes a series of projects related to the development of a fully functional text editor. This project describes how to replace a TextBox control with a RichTextBox control, which allows you to format different pieces of text in different ways. We consider the capabilities of the RichTextBox control related to formatting fonts and paragraphs. In addition, we describe tools that allow to determine the position of the cursor in the text and save the text in various formats.

17.1. Replacing the TextBox control with the RichTextBox control

The previously developed TEXTEDIT5 project (see Chapter 16) should be used as a template for the TEXTEDIT6 project.

In order to replace the textBox1 control of TextBox type already present in Form1 with a RichTextBox control (with richer formatting capabilities), it is enough to slightly correct the **Form1.Designer.cs** file. Recall that this file contains information that was added to the project while working with the form designer (including the menu designer) and the **Properties** window, so there is usually no need to explicitly edit it. However, nothing prevents you from making changes to the **Form1.Designer.cs** file.

Load the **Form1.Designer.cs** file into the editor (the easiest way to do this is using the **Solution Explorer** window), find the line in it

```
private System.Windows.Forms.TextBox textBox1;
```

and change it as follows:

```
private System.Windows.Forms.TextBoxRichTextBox textBox1;
```

Thus, we have changed the type of the textBox1 control. We will not change the *name* of the control, since it is used repeatedly in the existing code of our program.

We also need to change the statement that *creates* the textBox1 control. This statement is contained in the same **Form1.Designer.cs** file in the **Windows Form Designer generated code** section, which is hidden by default. Expand this section by clicking the + sign to the left of its header and find the statement

```
this.textBox1 = new System.Windows.Forms.TextBox();
```

To find the specified statement, you can, for example, organize a search for the word **textBox1** (for the search modes available in the Visual Studio environment, see Comment 2 in Section 9.5). Modify the found statement as follows:

```
this.textBox1 = new System.Windows.Forms.TextBoxRichTextBox();
```

After making these changes, the **Form1.Designer.cs** file can be closed by right-clicking on its tab header and choosing **Close** from the context menu that appears (or simply pressing Ctrl+F4).

The RichTextBox control allows you to set *different formatting settings* for different pieces of text. These settings are saved with the text in a special format called *Rich Text Format* (*RTF* format). Files containing text in this format usually have the .**rtf** extension. Let us modify the program to use the additional functionality of the RichTextBox control.

For the saveFileDialog1 and openFileDialog1 controls, change the DefaultExt property to **rtf** and the Filter property to **RTF files|* .rtf**.

Modify the SaveToFile and open1_Click methods of the Form1 class:

```
private void SaveToFile(string path)
{
  File.WriteAllText(path, textBox1.Text);
  textBox1.SaveFile(path, RichTextBoxStreamType.RichText);
  textBox1.Modified = false;
}
private void open1_Click(object sender, EventArgs e)
{
  if (TextSaved())
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      string path = openFileDialog1.FileName;
      textBox1.Text = File.ReadAllText(path);
      textBox1.LoadFile(path, RichTextBoxStreamType.RichText);
      Text = "Text Editor - " + Path.GetFileName(path);
      saveFileDialog1.FileName = path;
      openFileDialog1.FileName = "";
    }
}
```

In the bold1_Click and font1_Click methods, replace all **textBox1.Font** fragments with **textBox1.SelectionFont** (bold1_Click should have *three* such fragments, and font1_Click should have *four* fragments).

In the fontColor1_Click method, replace all **textBox1.ForeColor** fragments with **textBox1.SelectionColor** (there should be *two* such fragments).

In the backgroundColor1_Click method, replace all **textBox1.BackColor** fragments with **textBox1.SelectionBackColor** (there should be *two* such fragments).

In the leftJustify1_Click method, replace **textBox1.TextAlign** with **textBox1.SelectionAlignment** in the last statement.

Finally, set the EnableAutoDragDrop property of the textBox1 control to **True**.

Result. *Font formatting commands* now affect the selection or (if there is no selection in the text) subsequent characters entered. *Paragraph alignment commands* affect the selected paragraphs including paragraphs in which only part of the text is selected or (if no paragraphs are selected) the current paragraph. You can save text in a file with format settings (the default file extension is .**rtf**). When loading files in RTF format, the text is displayed on the screen with the format settings saved.

If the text does not fit in height in the editor area, a *vertical scroll bar* appears on the right (recall that the TextBox control does not have the same ability to automatically display a vertical scroll bar). Lines that are too wide are, as before, automatically *wrapped* to a new line (see Comment 5 in Section 12.3).

Thanks to the **True** value of the EnableAutoDragDrop property, the editor provides a *drag-and-drop mode* for selections: if you left-click on the selection, you can *move* it to a new location; if you hold down the Ctrl key while dragging (in this case, a + sign is displayed near the mouse cursor), *copying* of the selection is performed instead of its moving.

Disadvantages. (1) When you move the cursor to an area of text with a *different* formatting, the state of the shortcut buttons and formatting items in the **Format** menu does not change. (2) The Modified attribute is not cleared when a new file is loaded. (3) When the **New** and **Open** commands are executed, the formatting settings in the menu and on the toolbar are not adjusted.

In addition to these easily identifiable disadvantages, the new version of our program contains an error that is not easy to find. In order to detect this error, you need to select a piece of text containing *more than one type of font* (for example, **Times New Roman** and **Arial** fonts) and try to execute one of the font style setting commands (**Bold**, **Italic**, or **Underline**) or set a new type of font for the selection using the **Font** command. In this situation, a NullReferenceException will be thrown.

All these disadvantages and errors will be corrected in the next section.

Comments

1. When changing the methods related to saving and loading files, the SaveFile and LoadFile methods of the RichTextBox control were used (the TextBox control does not have similar methods).

2. When changing methods related to formatting, the SelectionFont, SelectionColor, SelectionBackColor, and SelectionAlignment properties of the RichTextBox control were used. All these properties allow you to define and change the formatting options for the selection or (in the absence of selection) the formatting options for the position of the text at which the keyboard cursor (the *caret*) is. The first property is responsible for the characteristics of the font, the second for the color of the characters, the third for the background color, and the fourth for the horizontal alignment of the paragraph. Some other properties related to paragraph formatting will be used in Section 17.3.

17.2. Correcting the state of shortcut buttons and menu commands when changing the current format

Add a new SetEnabled method to the Form1 class:

```
private void SetEnabled(bool value)
{
  bold1.Enabled = bold10.Enabled =
    italic1.Enabled = italic10.Enabled =
    underline1.Enabled = underline10.Enabled =
    font1.Enabled = value;
}
```

Define an event handler for the SelectionChanged event for the textBox1 control:

textBox1.SelectionChanged handler

```
private void textBox1_SelectionChanged(object sender, EventArgs e)
{
  Font f = textBox1.SelectionFont;
  SetEnabled(f != null);
  if (f != null)
  {
    bold1.Checked = bold10.Checked = f.Bold;
    italic1.Checked = italic10.Checked = f.Italic;
    underline1.Checked = underline10.Checked = f.Underline;
  }
  ToolStripMenuItem mi = leftJustify1;
  switch (textBox1.SelectionAlignment)
  {
    case HorizontalAlignment.Center:
      mi = center1; break;
    case HorizontalAlignment.Right:
      mi = rightJustify1; break;
  }
  if (mi == alignItem) return;
  alignItem.Checked = GetButton(alignItem).Checked = false;
  mi.CheckState = CheckState.Indeterminate;
  GetButton(mi).Checked = true;
  alignItem = mi;
}
```

In the new1_Click method, after the statement

```
saveFileDialog1.FileName = "";
```

insert the following statement:

```
textBox1_SelectionChanged(this, null);
```

In the open1_Click method, after the statement

```
openFileDialog1.FileName = "";
```

insert the following statements:

```
textBox1.Modified = false;
textBox1_SelectionChanged(this, null);
```

Result. We have corrected all the disadvantages and errors noted in Section 17.1. In particular, the state of shortcut buttons and menu items now matches the format of the current text position.

If a fragment is selected in the text, the appearance of menu items and shortcut buttons depends on whether the selection contains parts with different formatting. If the selection contains *several types of fonts*, then the font setting menu items (**Bold**, **Italic**, **Underline**, and **Font**) and corresponding shortcut buttons become unavailable (this avoids the error noted in Section 17.1). If the selection contains one type of font, then the indicated menu and toolbar controls are available; they are in the checked state only if the *entire selection* has this font style (for example, the **Bold** shortcut button will be "pressed" only if the entire selection is in bold).

Commands related to text alignment behave in a similar way: if the same alignment method is set for the entire selection, then the menu item and shortcut button corresponding to this alignment method are in the checked state; if the selection contains paragraphs with different alignment options, the **Left justify** command and its associated shortcut button are in the checked state.

Comment

When using the SelectionFont property, it is necessary to remember about its important feature: if there is *more than one type of font* in the selection, the SelectionFont property is null. In this situation, attempting to access any member of the SelectionFont property (such as its Bold, Italic, or Underline properties) will throw a NullReferenceException. For this reason, in the textBox1_SelectionChanged method, the members of the SelectionFont property are accessed only if this property is not null. However, there are two more methods in our program that call the members of the SelectionFont property: these are new versions of the bold1_Click and font1_Click methods. To avoid a possible error when executing the bold1_Click and font1_Click methods, all menu items and shortcut buttons that call these methods are made *unavailable* if a fragment containing more than one type of font is selected in the text (for this, the SetEnabled helper method is called in the textBox1_SelectionChanged handler).

Note that there is no danger of an error for *text alignment* commands: if the selection contains several alignment options, the SelectionAlignment property returns the HorizontalAlignment.Left value. Thus, in such a situation, the menu item and the shortcut button corresponding to the left alignment will be in the checked state, which seems quite natural. There are also no problems with the

commands for setting the color of symbols and background (see Section 13.3): if, when executing these commands, there is a selection with different color options, then the black color will be highlighted in the color selection dialog box.

17.3. Setting paragraph properties

Add a new form named Form2 to the project and place a container control of GroupBox type on this form (this control will be named groupBox1). Place three labels (label1, label2, label3) and three NumericUpDown controls (numericUpDown1, numericUpDown2, numericUpDown3) in the groupBox1 control. Also, place another label (label4), a drop-down list (the ComboBox control named comboBox1), a checkbox (checkBox1), and two buttons (button1 and button2) on Form2.

Remark. The ComboBox control will be considered in more detail in the LISTBOXES project (see Chapter 19).

Configure the properties of Form2 and its controls and arrange the controls as shown in Fig. 17.1. Recall that setting Modifiers = **Internal** allows to refer to a control from another form of the same project.

Properties

```
Form2: Text = Paragraph, MaximizeBox = False,
  MinimizeBox = False, FormBorderStyle = FixedDialog,
  StartPosition = CenterScreen, ShowInTaskbar = False,
  AcceptButton = button1, CancelButton = button2
groupBox1: Text = Indentation
label1: Text = From left:
label2: Text = From right:
label3: Text = From bullet:, Enabled = False
label4: Text = Alignment:
numericUpDown1: Modifiers = Internal
numericUpDown2: Modifiers = Internal
numericUpDown3: Modifiers = Internal, Enabled = False
comboBox1: DropDownStyle = DropDownList, Modifiers = Internal
checkBox1: Text = Bulleted paragraph, Modifiers = Internal
button1: Text = OK, DialogResult = OK
button2: Text = Cancel
```

Additionally, define the Items property of the comboBox1 control. A special dialog box is provided for this property; this dialog box can be called from the **Properties** window by clicking the ellipsis button ... near the property text box. In our case, three alignment options must be input into this dialog box, one per line:

```
Left aligned
Right aligned
Centered
```

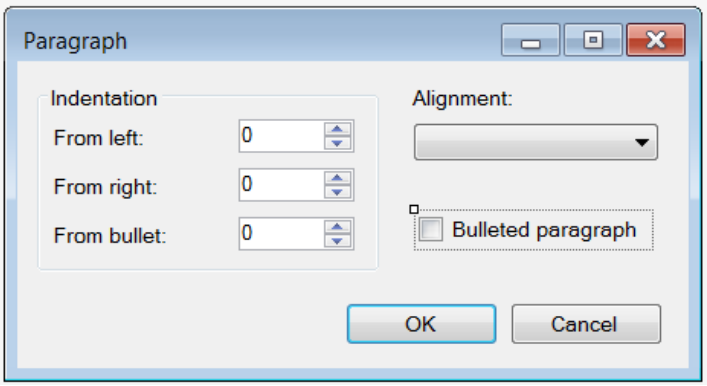


Figure: 17.1. Form2 view

Define a handler for the CheckedChanged event for the checkBox1 control:

checkBox1.Click handler (Form2)

```
private void checkBox1_Click(object sender, EventArgs e)
{
  label3.Enabled = numericUpDown3.Enabled = checkBox1.Checked;
  numericUpDown3.Value = numericUpDown3.Enabled ? 10 : 0;
}
```

At the beginning of the Form1 class declaration, add a declaration of the new field:

```
private Form2 form2 = new Form2();
```

In the constructor of the Form1 class, add the statement

```
AddOwnedForm(form2);
```

Add a menu item with text **&Paragraph...** to the drop-down menu associated with the **Format** menu item (see Sections 13.1–13.4). Set the Name property of the added menu item to paragraph1 and define the Click event handler for this menu item:

paragraph1.Click handler (Form1)

```
private void paragraph1_Click(object sender, EventArgs e)
{
  form2.checkBox1.Checked = textBox1.SelectionBullet;
  form2.comboBox1.SelectedIndex =
    (int)textBox1.SelectionAlignment;
  form2.numericUpDown1.Value = textBox1.SelectionIndent;
  form2.numericUpDown2.Value = textBox1.SelectionRightIndent;
  form2.numericUpDown3.Value = textBox1.BulletIndent;
  if (form2.ShowDialog() == DialogResult.OK)
  {
    textBox1.SelectionIndent = (int)form2.numericUpDown1.Value;
    textBox1.SelectionRightIndent =
      (int)form2.numericUpDown2.Value;
    textBox1.BulletIndent = (int)form2.numericUpDown3.Value;
```

```
    textBox1.SelectionBullet = form2.checkBox1.Checked;
    textBox1.SelectionAlignment =
      (HorizontalAlignment)form2.comboBox1.SelectedIndex;
    textBox1_SelectionChanged(this, null);
  }
}
```

Result. New menu item **Format | Paragraph...** allows you to customize the properties of the current paragraph or a group of selected paragraphs. During its execution, the **Paragraph** dialog box is displayed, in which you can specify the amount of indentation of the paragraph from the left and right bounds of the editing area (in pixels), as well as the type of alignment. In addition, by selecting the **Bulleted paragraph** checkbox, you can add a *bullet* in the form of a black marker (•) to the paragraph; in this case, you can specify the amount of indentation from the bullet to the text.

Comments

1. In this section, another group of properties of the RichTextBox control was used, which is related to the paragraph settings:

- SelectionIndent and SelectionRightIndent set the indentation in pixels from the left and right bounds of the editing area, respectively;
- SelectionBullet is boolean; if it is true, then the paragraph is labeled with the *bullet*, that is, a special marker •;
- BulletIndent sets the indent in pixels from the bullet to the text following it.

2. The order of items in the comboBox1 drop-down list corresponds to the order in which the alignment options are specified in the HorizontalAlignment enumeration: Left (0), Right (1), Center (2). Due to this circumstance, to determine the selected alignment option, it is enough to convert the SelectedIndex property of the comboBox1 control, which contains the number of the selected list item, to the HorizontalAlignment type (see the last but one statement in the paragraph1_Click method). A subsequent call to the textBox1_SelectionChanged handler adjusts the state of alignment-related menu items and shortcut buttons.

17.4. Display the current row and column

Proceeding in the same way as in Section 16.1, add another (fifth) label called position1 to the statusStrip1 status bar and set the properties of the added label:

Properties

```
position1: Text = 1 : 1, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 60
```

Press the left mouse button on the position1 label and move it to the left of the hint1 label (with the text **Ready**). As a result, the indicated labels will be swapped (see Fig. 17.2).

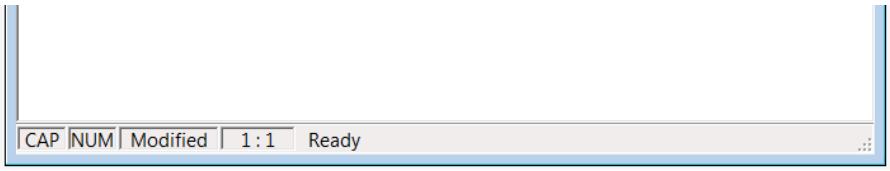


Fig. 17.2. The lower part of Form1

Add a new piece of code to the beginning of the textBox1_SelectionChanged method:

```
int x = textBox1.SelectionStart,
  y = textBox1.GetLineFromCharIndex(x),
  x0 = textBox1.GetFirstCharIndexFromtLine(y);
position1.Text = string.Format("{0} : {1}", y + 1, x - x0 + 1);
```

Result. The status bar displays the *current position* of the keyboard cursor (the caret) in the format row : column, where row is the line number and column is the character number in the line (lines and characters are numbered from 1). If the edited text contains a selection, then the position of the beginning of this selection is indicated.

Comment

To determine the current position of the keyboard cursor in the text of the textBox1 control (this text is contained in the Text property of this control), the following properties and methods are used that are available for both the TextBox control and the RichTextBox control:

- the previously mentioned SelectionStart property (see Comment 2 in Section 8.1), which allows you to determine the position of the cursor or the beginning of the selection in the Text string;
- the GetLineFromCharIndex(n) method, which allows you to determine the line number for the character with the number n in the Text string property;
- the GetFirstCharIndexFromLine(n) method, which allows you to determine the first character number of the line with number n in the Text property.

Both strings and characters are numbered from 0, so we add 1 to the resulting values.

Instead of the GetFirstCharIndexFromLine(n) method, we can use the GetFirstCharIndexOfCurrentLine method (without parameters), which allows us to determine the character number from which the current line (that is, the line containing the keyboard cursor) begins.

Note that the line-separated content of the TextBox and RichTextBox controls can be accessed using the Lines property, which is an array of strings.

Also note one more method related to the position of the keyboard cursor: ScrollToCaret. This method, which has no parameters, allows you to display on the screen a piece of text containing the keyboard cursor. If the keyboard cursor is already on the screen when the method is called, the ScrollToCaret method does nothing.

17.5. Loading and saving text without format settings

For the saveFileDialog1 and openFileDialog1 controls, change the values of the Filter property to **RTF files|* .rtf|Text files|* .txt|All files|*. ***.

As a result, *three* filters are associated with the specified dialog boxes: for *RTF files* (with the .**rtf** extension), for *plain text files* (with the .**txt** extension), and for *arbitrary files* with any valid name and extension.

Add a new GetFileType method to the Form1 class:

```
private RichTextBoxStreamType GetFileType(string path)
{
  string s = Path.GetExtension(path).ToUpper();
  return s == ".RTF" ? RichTextBoxStreamType.RichText :
    RichTextBoxStreamType.PlainText;
}
```

Modify the SaveToFile and open1_Click methods:

```
private void SaveToFile(string path)
{
  textBox1.SaveFile(path, RichTextBoxStreamType.RichText);
  textBox1.SaveFile(path, GetFileType(path));
  textBox1.Modified = false;
}
private void open1_Click(object sender, EventArgs e)
{
  if (TextSaved())
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      string path = openFileDialog1.FileName;
      textBox1.LoadFile(path, RichTextBoxStreamType.RichText);
      textBox1.LoadFile(path, GetFileType(path));
      Text = "Text Editor - " + Path.GetFileName(path);
      saveFileDialog1.FileName = path;
      openFileDialog1.FileName = "";
    }
}
```

Result. When saving a document in a file with any extension other than .**rtf**, only *plain text* (without format settings) is written to the file in *ANSI encoding*, that is, in the one-byte encoding used by Windows by default. However, in the editor itself, the format settings are preserved and the formatted text can be saved later in a file with the .**rtf** extension. It is now possible to load into the editor both files in RTF format (with the .**rtf** extension) and plain text files. In order to speed up the selection of files with the .**rtf** and .**txt** extensions, as well as to be able to select files with any extension, the dialog boxes for opening and

saving files provide appropriate filters (**RTF files**, **Text files**, **All files**) listed in the **File type** drop-down list.

Here is a screenshot of a running program (Fig. 17.3). The keyboard cursor is positioned on the first centered line.

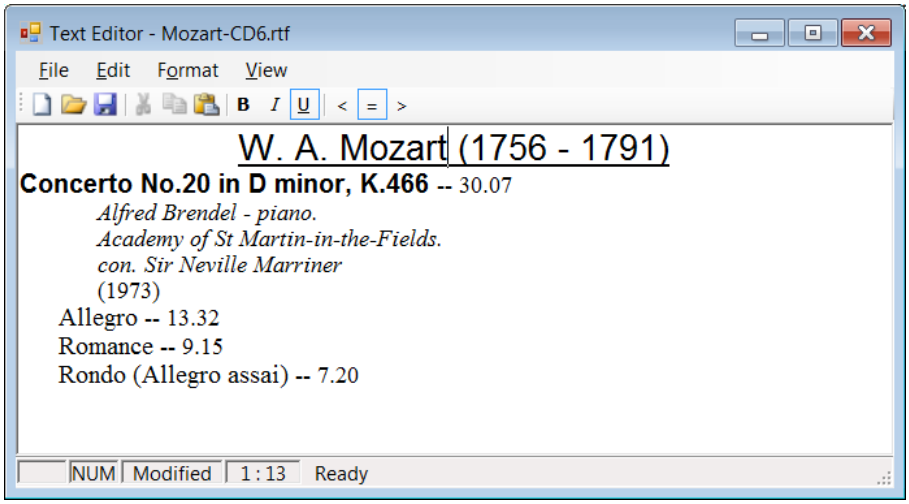


Figure: 17.3. View of the running TEXTEDIT6 application

Comments

1. To extract the extension from the file name (see the GetFileType method), the GetExtension method of the Path class is used, which returns the file extension along with the preceding dot (.). If the filename does not contain an extension or ends in a dot, an empty string is returned. See also Comment 2 in Section 18.2 about using the ToUpper method.

2. The SaveFile and LoadFile methods of the RichTextBox control allow saving and loading data in various formats. The format is determined by the second parameter of these methods, which is of the RichTextBoxStreamType enumerated type. The new version of the program uses not only the RichText format option (for RTF files), but also the PlainText format option, which allows you to save text without format settings in the standard Windows ANSI encoding. Among other formats, we note UnicodePlainText, which allows you to save text without format settings in Unicode encoding.

3. When working with the saveFileDialog1 dialog box for saving file, the user may experience certain inconveniences. For example, if the user created a file **test.rtf** and now wants to save it in the plain text format, then he/she will need to delete the .**rtf** extension in the **File name** text box of the save dialog box and explicitly input the .**txt** extension. In such situations, it is more natural to change the file type using the **File type** drop-down list. However, changing the file type in this list does not automatically change the file extension in the **File name** text box. The required change could have been provided in the event handler associated with file type changing, but, unfortunately, such an event is missing in the SaveFileDialog and OpenFileDialog controls.

18. Colors: COLORS project

The COLORS project introduces classes for working with color (Color, KnownColor, ColorTranslator) and controls that provide data scrolling (TrackBar, HScrollBar, VScrollBar). It also describes how to access controls through shortcut keys for their associated labels and discusses options for anchoring controls to boundaries of the form.

18.1. Defining a color as a combination of four color components. Track bars and scroll bars

After creating the COLORS project, place the TrackBar control (named trackBar1) on Form1 and set the properties for the form and this control:

Properties

```
Form1: Text = Colors, StartPosition = CenterScreen
trackBar1: LargeChange = 32, Maximum = 255,
  TickFrequency = 32, TickStyle = Both
```

With trackBar1 still selected, press Ctrl+C (one time) and then Ctrl+V (four times). As a result, four more TrackBar controls named trackBar2 – trackBar5 will be added to the form, and, for all these controls, all the property values (with the exception of Name and Location) will be the same as values of the corresponding properties of the trackBar1 control (note that such copying of controls can also be performed using the form's context menu).

For the trackBar1 control, additionally set the Value property to **255** (for other TrackBar controls, this property should remain equal to **0**). Place all controls on the form from top to bottom (see Fig. 18.1).

After that, place five labels (label1 – label5) on the form and set their Text properties to **Alpha**, **Red**, **Green**, **Blue**, **Gray**, respectively.

The labels must be vertically aligned to the middle of the corresponding TrackBar controls (see Fig. 18.1). For this, it is convenient to use the **Layout** panel (recall that it can be displayed on the screen using the **View | Toolbars | Layout** menu command). To perform alignment, first click on one of the TrackBar controls, then click on the corresponding label while pressing the Ctrl key, and then click on the **Align Middles** button  on the **Layout** panel. The order in which the controls are selected is significant, since the alignment is performed on the active control (its markers are white).

Finally, place the Panel container control (named panel1) at the bottom of the form and place another label (named label6) in this control. Set the properties for label6 as follows (the easiest way to set values for the ForeColor and BackColor

properties is to type these values on the keyboard without using the drop-down list):

Properties

```
label6: Text = Color, AutoSize = False, Dock = Fill,
  ForeColor = White, BackColor = Black
```

The final view of the form at this stage of its development is shown in Fig. 18.1.

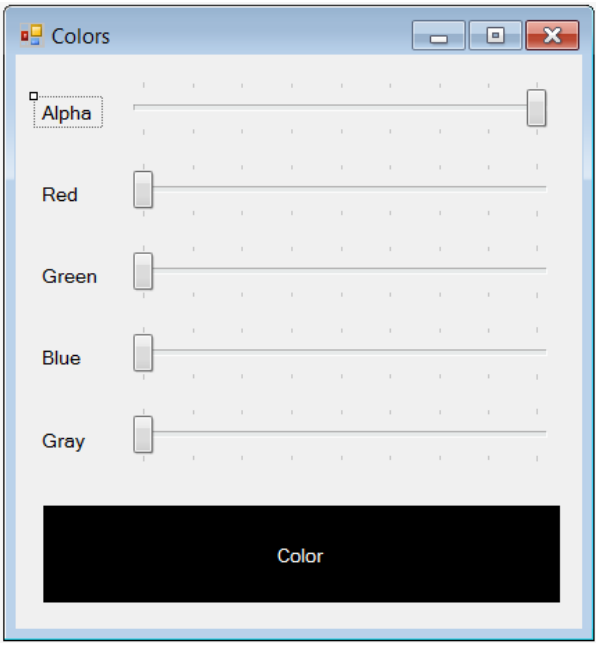


Fig. 18.1. Form1 view at the initial stage of development

Define the Scroll event handler for trackBar1, then connect this handler to the Scroll events of trackBar2 – trackBar4 (the Scroll event handler for trackBar5 will be added later, in Section 18.3):

trackBar1.Scroll handler

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value,
    trackBar2.Value, trackBar3.Value, trackBar4.Value);
}
```

For the panel1 control, set a *background image*. A good option is the **Chrysanthemum.jpg** file located in the Windows 7 system image directory **Users\Public\Pictures\Sample Pictures**. Windows 10 does not include this file, but it can be downloaded from the archive.org web-site using the link https://archive.org/details/Chrysanthemum_20160913.

To set the **Chrysanthemum.jpg** file as the background image for the panel1 control, first select this control in the form (since the panel1 control is under label6, the easiest way is to select this label and then press the Esc key). Then select the BackgroundImage property of the panel1 control in the **Properties** win-

dow and click the ellipsis button ... . In the **Select Resource** window that appears, do the following steps to load the required image (recall that these steps were previously described in Section 15.1): press the **Import** button (if the button is unavailable, then first select the **Project resource file** radio button); in the new **Open** window, select the required image file and then click the **Open** button or press the Enter key. After performing the described actions, the name of the selected file will appear in the list of resources and will be highlighted; in addition, an image taken from this file will appear on the right-hand side of the **Select Resource** window. It remains to close the **Select Resource** window by clicking **OK**. As a result, the BackgroundImage property of panel1 will be set to **COLORS.Properties.Resources.Chrysanthemum**. In addition, a new item **Chrysanthemum.jpg** will appear in the **Resources** section of the **Solution Explorer** window. It should be noted that the background image of panel1 is overlapped with label6, so we cannot see it on the form yet.

Use *non-aspect ratio scaling* for the background image of panel1 by setting the BackgroundImageLayout property to **Stretch** (see Section 21.4 for more information on image view modes).

Result. The background color of label6 is defined as a combination of four color components, namely transparency (*Alpha*) and the intensity of the three base colors: *Red*, *Green*, and *Blue*. Each color component can vary from 0 to 255; a value of 255 for the Alpha component corresponds to full opacity. In our program, the values of the color components are set by the position of the four corresponding TrackBar controls (the trackBar5 control is not used yet). Because the panel under label6 contains a background image, this image can be seen if a transparency level Alpha is less than 255.

Comments

1. Although the Visual Studio form designer provides a convenient way to align controls relative to each other (as well as relative to form boundaries) by simple dragging and dropping controls across the form, the **Layout** panel is very useful in some cases. In particular, it allows you to easily align groups of controls, increase proportionally the horizontal or vertical spacing between controls, set the selected controls to the same width or height, change the *z*-coordinate of a control by moving it “up” or “down” (for *z*-order, see Comment 4 in Section 9.1).

You can add to this panel new buttons by clicking the **Add or Remove Buttons** item on the right-hand side of this panel. From the drop-down list of buttons that appears, you can select, for instance, the buttons for centering control or grouping controls horisontally or vertically in the form, as well as the shortcut button for the **Tab Order** menu item (for this menu item, see Comment 1 in Section 8.1).

2. The TrackBar control (a *track bar*, or a *slider*) is convenient to use in situations when you need to set a parameter that accepts integer values from

a certain (not too large) range. When configuring the trackBar1 control in our project, we set the values of the following properties: LargeChange (step when pressing the PgUp and PgDn keys), Maximum (the maximum allowable value), TickFrequency (the delta between tick marks drawn), TickStyle (the view of the tick marks), and Value (the current value of the track bar). There was no need to change the other properties, as we can use their default values. Let us list some of these properties: SmallChange (when pressing the arrow keys, by default is 1), Minimum (the minimum allowable value, by default is 0), and Orientation (determines the orientation of the track bar, the default is **Horizontal**). Note that changing the slider by the LargeChange value is performed not only when you press PgUp or PgDn, but also when you click on a control to the left or right of a *slider* that marks the current value of the track bar (a slider is also called a *thumb* or *scroll box*).

3. The TrackBar control has one drawback: when changing its "thickness" (that is, the vertical size Height in the case of a horizontal orientation or the horizontal size Width in the case of a vertical orientation), the track bar elements are not centered relative to its new borders, and the track bar size is not proportionally changed (note that the thickness of the track bar can only be changed when the AutoSize property is set to **False**). This makes it impossible to use track bars of a small thickness.

If the default track bar thickness is not suitable, you can use alternative controls with similar properties: horizontal and vertical *scroll bars* HScrollBar and VScrollBar. In doing so, however, two important points should be taken into account.

First, by default, the scroll bar cannot receive focus (even when the mouse is clicked on it). If this behavior is undesirable, then you should set the value of its TabStop property to **True**.

Second, the maximum value that the Value property of the scroll bar can take is Maximum – LargeScroll + 1, that is, it can be *less* than Maximum. Such a strange, at first glance, behavior turns out to be actually quite natural, since the LargeScroll value for the scroll bar determines not only the step when pressing PgUp or PgDn, but also the size (width) of the slider relative the entire scroll bar, and the Value property corresponds to the position of the slider *border* (more precisely, the left border for a horizontal scroll bar and the top border for a vertical one). Therefore, if, for example, the Minimum property is 1, the Maximum property is 10, and the LargeScroll is 5, then the slider will have a width that is half the width of the scroll area, and, therefore, when it is moved to the right (or down for a vertical scroll bar), its left (respectively, upper) border will be located only at the value 6. This value will be the *maximum possible value* of the Value property (see the above formula). Note that this behavior is well suited for scrolling through text information using a vertical scroll bar, if we assume that the Value property corresponds to the number of the *top line* of

text displayed on the screen. In the example above, with Value = 6, the last 5 lines of the text will be displayed on the screen (from 6 to 10) and further scrolling down is no longer required.

If scroll bars were used in our example, then, to provide a range of 0–255 for the Value property while LargeScroll = 32, the Maximum property would have to be set to 286 (that is, 255 + LargeScroll – 1). Of course, all such problems are removed if we set LargeScroll equal to 1, however, in this case, the ability to iterate over values with a large step (for example, using the PgUp and PgDn keys) will be lost.

Concluding a brief overview of the features of scroll bars, we note that, using their Scroll event handler, you can very flexibly respond to any user actions related to the scroll bar (allowing, for example, not to handle each change in the Value property if the user drags the slider with the mouse, but react only to the final value of the Value property at the moment of releasing the slider). The Scroll event of the TrackBar control does not have such capabilities.

18.2. Inverting colors and output color constants

Add new statements to the trackBar1_Scroll method:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value,
    trackBar2.Value, trackBar3.Value, trackBar4.Value);
  Color c = label6.BackColor;
  label6.ForeColor = Color.FromArgb(0xFF ^ c.R, 0xFF ^ c.G,
    0xFF ^ c.B);
  label6.Text = c.Name.ToUpper();
}
```

Result. The numerical value of the current color in ARGB format (Alpha – Red – Green – Blue) is displayed on the panel as a hexadecimal number. In this case, two characters are used for each color component, and the letters A – F (corresponding to hexadecimal digits from 10 to 15) are displayed in upper case. For example, the color value for maroon (an opaque deep red color of intensity 128) is FF800000. Text color is opaque and *inverse* to the panel background color. See also Comments 1 and 2.

Disadvantage 1. When the program starts, label6 contains the text **Color**, not a numeric value of opaque black color.

Correction. Add the following statement to the constructor of the Form1 class:

```
trackBar1_Scroll(null, null);
```

Result. Now the trackBar1_Scroll method is called at the time of form creation, which ensures the correct setting of the label6 text. When calling the meth-

od, both of its parameters can be set to null, since none of the parameters are used in this method.

Remark. The noted disadvantage could be corrected simply by setting the value of the Text property of label6 to **FF000000** (a numeric value of opaque black color) in the **Properties** window. However, the used method of correction is more flexible, since it allows the initial text of the label to be displayed correctly after *any changes* to the trackBar1_Scroll method that can be made later (see Section 18.4).

Disadvantage 2. If transparency is less than 16, label6 displays less than 8 digits (in particular, with completely transparent black, the label will contain a *single* digit 0). This way of representing color is inconvenient; it is more natural to always display a hexadecimal number with 8 characters (two characters for each color component).

Correction. In the trackBar1_Scroll method, *replace* the last statement as follows:

```
label6.Text = c.Name.ToUpper();
label6.Text = c.ToArgb().ToString("X8");
```

Result. Now the number c.ToArgb() is displayed in *hexadecimal format* X with capital Latin letters and always contains 8 characters (we used the formatting version of the ToString method for the int type).

Comments

1. Read-only properties A, R, G, B of the Color class allow you to get the numerical value of the corresponding color component. To invert each of the base colors, the bitwise operator ^ (exclusive OR) is used. When using a version of the FromArgb method with three parameters (R, G, B), the transparency A is assumed to be 255.

2. The ToUpper method of the string class converts all alphabetic characters in the string to uppercase (the reverse method is named ToLower). Characters that are not letters are not modified. The ToUpper and ToLower methods correctly process not only Latin alphabet, but also letters of other alphabets.

18.3. Grayscale colors

Define the Scroll event handler for the trackBar5 control:

trackBar5.Scroll handler

```
private void trackBar5_Scroll(object sender, EventArgs e)
{
  trackBar2.Value = trackBar3.Value =
    trackBar4.Value = trackBar5.Value;
}
```

Result. Moving the trackBar5 slider causes all three base colors to change synchronously resulting in different *grayscale* colors (the transparency value does not change).

Error. Although moving the trackBar5 slider changes the trackBar2 — trackBar4 sliders synchronously, this change does not affect the color of label6. This error is due to the fact that, when the Value property is programmatically changed, the Scroll event is not raised.

Correction. Add the following statement to the trackBar5_Scroll method:

```
trackBar1_Scroll(null, null);
```

Remark. If the program must react not only to changes in the position of the slider made by the user, but also to *any* changes of the Value property, then we can use the ValueChanged event handler. For example, if we had defined the ValueChanged event handler for the trackBar1 — trackBar4 controls instead of the Scroll event handler, then the first version of the trackBar5_Scroll method would work correctly.

It should be noted, however, that the ValueChanged event handler will be called much more often than the Scroll event handler. In particular, when the trackBar5_Scroll method is executed, this handler will be called 3 times.

18.4. Displaying color names

Modify the trackBar1_Scroll method as follows:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value,
    trackBar2.Value, trackBar3.Value, trackBar4.Value);
  Color c = label6.BackColor;
  label6.ForeColor = Color.FromArgb(0xFF ^ c.R,
    0xFF ^ c.G, 0xFF ^ c.B);
  label6.Text = c.ToArgb().ToString("X8");
  string s = c.ToArgb().ToString("X8");
  switch (c.A)
  {
    case 0:
      s += " Transparent";
      break;
    case 255:
      string
        s0 = ColorTranslator
          .FromWin32(ColorTranslator.ToWin32(c)).Name;
      if (s0.Substring(0, 2) != "ff")
        s += " " + s0;
      break;
  }
  label6.Text = s;
```

```
}
```

Result. In the case when the current color is associated with a certain *name* (for example, **Black** or **Maroon**), the panel displays not only the numerical value of the current color in hexadecimal format, but also its name. If transparency has a zero value, then the text **Transparent** is displayed next to the numerical value of the color.

Comments

1. All colors that have names (the *named colors*) are contained in the KnownColor enumeration. The Color structure has the ToKnownColor method that returns a corresponding KnownColor enumeration member for any named color or 0 if the color is not a named color. However, the usefulness of the ToKnownColor method is significantly limited by the fact that, if a color was created using the FromArgb method, then calling the ToKnownColor method for it will necessarily return 0 (even if the color with the specified components is included in the set of named colors). This behavior is due to the following feature of the Color structure: when analyzing (in particular, comparing) structures of Color type, not only their color components are taken into account, but also the way these structures were created. For example, if the structure c1 of Color type was created using the FromArgb(0, 0, 0) method and the structure c2 was created using the FromName("Black") method, then they will be considered different, although both represent opaque black color (note also that the expression c1.Name will return **ff000000** and the expression c2.Name will return **Black**).

2. Since the direct using the ToKnownColor method in our case will not allow obtaining the required result, it would be possible to form an array of all named colors, using the KnownColor enumeration for this (see Section 19.1), and then compare the color characteristics of each named color with the color characteristics of the color of interest. However, .NET library includes the ColorTranslator class that makes it easier to recognize a named color. If you create a color using the FromWin32 method of the ColorTranslator class, then, in the case of a named color, its Name property will return the name of the color; otherwise, Name will return the numeric representation of the color in ARGB format. To be able to apply the FromWin32 method to an object of Color type, this object must first be converted to the integer RGB format using the ToWin32 method. Note that, when using the methods of the ColorTranslator class, transparency will not be taken into account, since the color format for Win32 (RGB format) does not provide such a color component. Therefore, we can use the methods of the ColorTranslator class *only* for opaque colors, that is, colors with Alpha = 255. This approach is perfectly valid, since *all* named colors, except for the fully transparent color named **Transparent**, are opaque. In addition, we handle the situation of full transparency Alpha = 0 in a special way.

3. Checking s0.Substring(0, 2) != "ff" allows us to determine if string s0 contains a color name or a numeric representation of a color. In the latter case, the

string s0 always starts with two **ff** characters, corresponding to Alpha = 255 (fully opaque). To get a substring of string s0, the Substring(start, len) method of the string class is used, which returns a substring of length len starting at the character with index start. There is a version of this method with one start parameter; this version returns the rest of the string, starting at the start character.

18.5. Controls and their associated labels

Modify the Text properties for labels label1 – label5 by adding the & character: **&Alpha**, **&Red**, **&Green**, **&Blue**, **Gra&y** (see Fig. 18.2). For a **Gray** label, the & character selects the *last* letter, since all previous letters have already been used as selected characters in other labels.

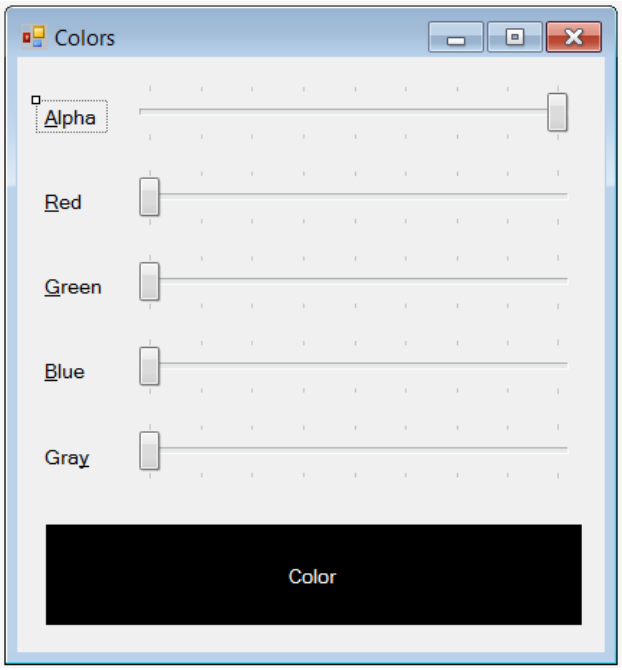


Fig. 18.2. The final view of Form1

Using the **View | Tab Order** menu command (see Comment 1 in Section 8.1), set the TabIndex property values for controls placed on the form as follows: label1 – **0**, trackBar1 – **1**, label2 – **2**, trackBar2 – **3**, label3 – **4**, trackBar3 – **5**, label4 – **6**, trackBar4 – **7**, label5 – **8**, trackBar5 – **9**.

Result. Switching between track bars can now be done using Alt-combinations of characters underlined in the labels to the track bars (Alt+A for the track bar that determines the transparency, Alt+R for the track bar that determines the intensity of the red color, etc.).

Remark. If, when starting the program, there are no underlined letters in the labels, then press the Alt key. In addition, the current track bar may not have a selection frame. To display the selection frame, press the Tab key.

Comment

If the Alt-combination is assigned to a control that cannot receive focus (for example, the Label control), then, when such a combination is pressed,

an attempt is made to set focus on the *next* control (that is, on the control with a greater TabIndex property value). If several controls have the same TabIndex property value, then they are selected in *z*-order, that is, in ascending *z*-coordinate (for *z*-order, see Comment 4 in Section 9.1).

18.6. Anchoring controls

Change the Anchor property for the trackBar1 – trackBar5 controls by setting it to **Top, Left, Right**. When setting the Anchor property in the **Properties** window, a special panel is displayed. To configure the required option in this panel, select the line leading to the *right* border of the panel, since the lines leading to the top and left border are already highlighted by default).

Using similar actions, set the Anchor property of the panel1 control equal to **Top, Bottom, Left, Right** (in this case, in the Anchor property settings panel, select the lines leading to the *bottom* and *right* borders, leaving also selected lines leading to the top and left borders). We emphasize that it is necessary to configure the Anchor property of panel1, *not* label6 which is contained in the panel.

Result. If you change the size of Form1 during program execution, the size of the controls will be adjusted in accordance with the new size of the form: the trackBar1 – trackBar5 track bars change the width, the panel1 panel change the width and height (see Fig. 18.3). See also Comment 1.

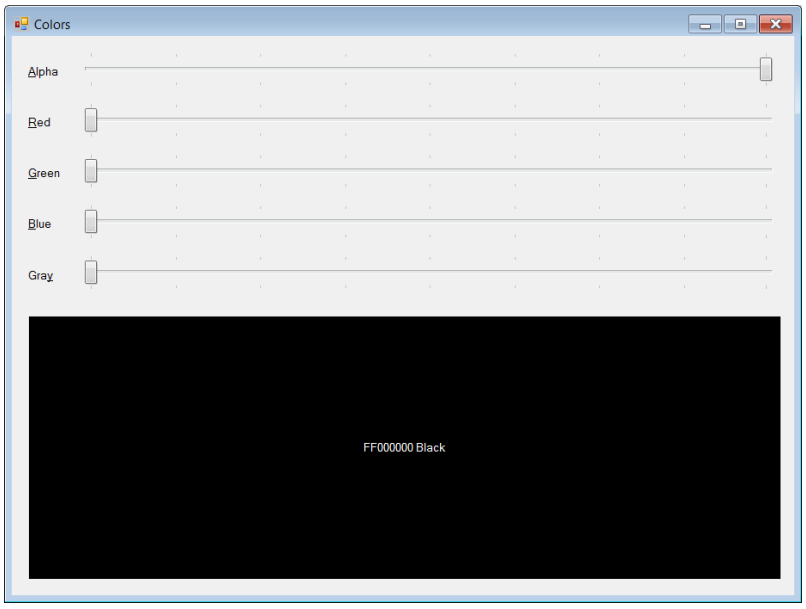


Fig. 18.3. View of the running COLORS application after increasing the size of the form

Disadvantage 1. When you reduce the size of the form, a situation may arise when panel1 is no longer visible, and the track bars become too narrow (see Fig. 18.4).

Correction. Add the following statement to the constructor of the Form1 class:

```
MinimumSize = Size;
```

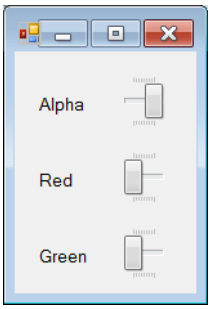


Fig. 18.4. View of the running COLORS application after decreasing the size of the form

Result. Now the size of the form can only be increased (compared to the initial one), since the minimum allowable size of the form, determined by the MinimumSize property, matches the initial size of the form stored in the Size property (see Comment 2).

Disadvantage 2. If you increase the size of the form (the size of panel1 will increase accordingly), then, when you change the position of the track bars, flickering occurs on the panel (in particular, the background image appears for a while, even in the case of completely opaque colors). It is interesting to note that flickering does not occur on the part of the panel that matches its initial size.

Correction. Change the Dock property of label6 to **None** and define the Resize event handler for Form1:

Form1.Resize handler

```
private void Form1_Resize(object sender, EventArgs e)
{
  label6.Size = panel1.Size;
}
```

Result. Now there is no flickering at any size of the form (see Comment 3).

Comments

1. The Anchor property is responsible for anchoring of visual controls. For any control, by means of this property, you can specify the borders of the form (or rather, the parent control) to anchor. By default, most controls are anchored to the top and left borders. When the form is resized, the distance from its "anchored" borders to the bounds of the control remains unchanged.

If the control is not anchored to any of the vertical borders, then, when the form is resized, the width of the control will not change and its distance to the left and right borders will change synchronously, ensuring that its relative position with respect to these borders is preserved. A similar effect is achieved when the control is not anchored to any of the horizontal borders.

Let us give an example. If you place a control in the center of the form and set anchoring to *all* borders, then whenever the form is resized, the control will remain in the center of the form and its size will change. If, however, anchoring is completely removed for such a control (that is, if its Anchor property is

set to **None**), then the control will still remain in the center of the form, but its size will not change.

Another way to keep the size and position of controls in sync with respect to the form is *docking*, which can be set using the Dock property (docking is discussed in detail in Section 21.3). These methods are mutually exclusive: if you change the default value for an Anchor or Dock property, the other property will automatically change to the default.

2. In addition to the MinimumSize property used to correct the diadvantage 1, there is a similar MaximumSize property that allows you to limit the maximum size of the form. If you do not need to restrict the size for some dimension, then it is enough to set the corresponding coordinate of the MinimumSize and MaximumSize properties to 0. Note that all visual controls also have these properties; by default, each of them is **0; 0**, that is, there is no size limitation.

3. The disadvantage 2 is apparently connected with the not quite correct implementation of the Fill docking mode, which we used to make label6 completely overlap the client area of panel1. To correct this, we turned off docking mode; instead, we explicitly resize label1 to fit the panel using the form's Resize event handler, which is executed whenever the form is resized.

19. Drop-down list and list box: LISTBOXES project

The LISTBOXES project is related to the ComboBox and ListBox controls that implement the *combo box*, *drop-down list*, and *list box*. Methods for performing standard actions on list items (adding, deleting, inserting, changing the order, etc.) are considered, and actions are described that allow you to move list items in drag-and-drop mode. The project also covers the KnownColor enumeration and the Beep method of the Console class.

19.1. Creating and using drop-down lists

After creating the LISTBOX1 project, place the comboBox1 and panel1 controls on Form1 (Fig. 19.1). Set the properties of the form and the added controls:

Properties

```
Form1: Text = ComboBox and ListBox, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
comboBox1: Text = empty string, DropDownStyle = DropDownList
panel1: BorderStyle = Fixed3D
```

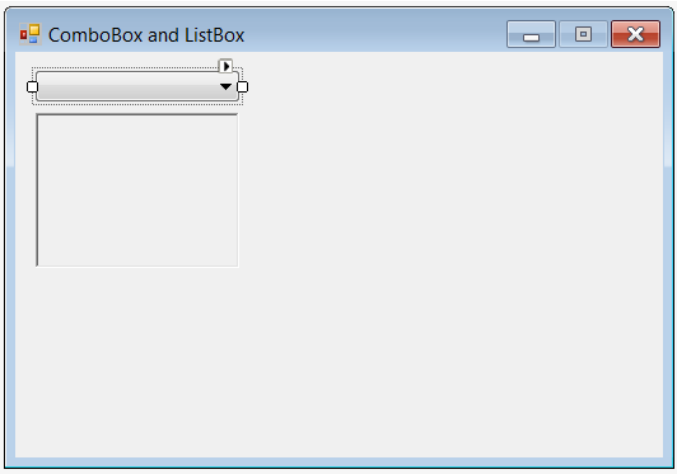


Fig. 19.1. Form1 view at the initial stage of development

Note that, by setting the DropDownStyle property of the comboBox1 control equal to **DropDownList**, we thereby disabled the ability to input new values for this control (only values from the existing list are allowed to be selected).

Thus, the ComboBox control allows you to create not only ordinary *drop-down lists*, but also drop-down lists that allow text input (such a control is called a *combo box*, since it combines the properties of lists and text boxes). For combo boxes, the DropDownStyle property must be set to **DropDown**.

Add new statements to the constructor for the Form1 class:

```
public Form1()
{
  InitializeComponent();
  string[] s = Enum.GetNames(typeof(KnownColor));
  int n1 = Array.IndexOf(s, "AliceBlue"),
    cnt = Array.IndexOf(s, "YellowGreen") - n1 + 1;
  string[] s0 = new string[cnt];
  Array.Copy(s,n1,s0,0,cnt);
  comboBox1.Items.AddRange(s0);
  comboBox1.SelectedIndex = 0;
}
```

Define an event handler for the SelectedIndexChanged event for the comboBox1 control:

comboBox1.SelectedIndexChanged handler

```
private void comboBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  panel1.BackColor = Color.FromName(comboBox1
    .Items[comboBox1.SelectedIndex].ToString());
}
```

Result. When the program starts, the drop-down list contains names of all *named colors* (see Section 18.4, Comment 1). The names are listed alphabetically. Selecting a name from the list changes the color of panel1 accordingly.

Comments

1. When forming a list of all named colors, the KnownColor enumeration was used. This enumeration starts with the names of the *system colors* (that is, the colors associated with various Windows elements; this group consists of 26 names), then there is the **Transparent** color (which is defined as transparent white), and then the names of the regular colors follow in alphabetical order (from **AliceBlue** to **YellowGreen**; the number of named regular colors is 140). In .NET 2.0, seven more system colors were added to the *end* of the KnownColor enumeration.

We can get an array of all names in the enumeration using the GetNames method. The Copy method of the Array class is used to select a range that contains only names of regular colors. The color indices of the AliceBlue and YellowGreen color names are determined using the IndexOf method of the Array class (the explicit constants 28 and 167 could have been used, but this would make the program less understandable).

2. To add the next item to the drop-down list, we can use the Add method of the Items collection property of the ComboBox control. However, if we want to

immediately add *all* elements from some array of strings to the list (as in our case), it is more convenient and faster to use the AddRange method.

3. Note that the items of the Items collection are described as object, so you need to use the ToString method to get their string representation.

19.2. List box: adding and removing items

Place listBox1, panel2, two labels label1 and label2 on Form1 (the panel2 control should be placed under the panel1 control, as shown in Figure 19.2). Set the properties of the added controls:

Properties

```
panel2: BorderStyle = Fixed3D
label1: Text = Add, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label2: Text = Delete, AutoSize = False,
  TextAlign = MiddleCenter, BorderStyle = FixedSingle
```

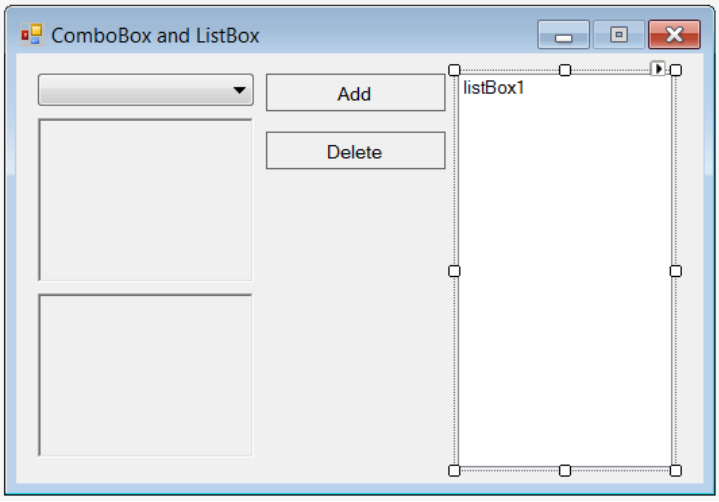


Fig. 25.2. Form1 view at the intermediate stage of development

Define handlers for the SelectedIndexChanged event for the listBox1 control and for the Click events for the label1 and label2 controls:

listBox1.SelectedIndexChanged, label1.Click, label2.Click handlers

```
private void listBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  panel2.Visible = listBox1.Items.Count > 0;
  if (listBox1.SelectedIndex == -1)
    return;
  panel2.BackColor = Color.FromName(listBox1
    .Items[listBox1.SelectedIndex].ToString());
}
private void label1_Click(object sender, EventArgs e)
```

```
{
  listBox1.SelectedIndex = listBox1.Items.Add(comboBox1.Text);
}
private void label2_Click(object sender, EventArgs e)
{
  listBox1.Items.RemoveAt(listBox1.SelectedIndex);
}
```

Result. When you click on the **Add** label, the selected color name from the comboBox1 drop-down list is added to listBox1 and also Panel2 color corresponds to the selected list item. If the list box is empty, panel2 is not displayed on the form. Clicking on the **Delete** label removes the selected item from the list box. See also Comments 1–2.

Disadvantage. After performing the delete operation, the list box does not contain any selected item (although the item surrounded by a frame remains). In this situation, the SelectedIndex property of the listBox1 control is –1, so clicking the **Delete** label again results in a runtime error (since –1 is not a valid value for the RemoveAt method). For the same reason, clicking on the **Delete** label in the case of an empty list box results in an error.

Correction. Change the label2_Click method as follows:

```
private void label2_Click(object sender, EventArgs e)
{
  listBox1.Items.RemoveAt(listBox1.SelectedIndex);
  int i = listBox1.SelectedIndex;
  if (i == -1)
  {
    Console.Beep();
    return;
  }
  listBox1.Items.RemoveAt(i);
  if (i == listBox1.Items.Count)
    i--;
  listBox1.SelectedIndex = i;
}
```

Result. When you delete an item in the middle of the list, the selection remains at its current position (which is now occupied by the next item). When you delete an item at the end of the list, the previous item is selected. Thus, if the list is not empty, it always has a selected item. If you click the **Delete** label when the list is empty, you will hear a beep (see Comment 3).

Remark. Note that in the listBox1_SelectedIndexChanged method, we have provided special handling of the situation SelectedIndex == –1, since this situation occurs when a list item is deleting.

Comments

1. The selected list item is displayed on a colored (usually blue) background. If the list has focus (that is, it is the active control of the form) and the Tab key has been used at least once to switch between form controls, then the selected item is additionally outlined with a dotted frame. An item surrounded by such a frame is called a *current item*.

In some interface libraries, you can distinguish between the current and selected list item. This is very useful when a list can contain multiple selected items. However, although the ListBox control in the **Windows Forms** library has a SelectionMode property that allows you to set the *multiple selection mode*, the above mentioned ability to determine the current list item is *not* provided in this library. In other words, if several items are selected in the list, then you can find out by software means which items are selected (that is, highlighted), but there are no easy way to determine which item is current (that is, outlined with a dotted frame). This circumstance makes it difficult to work with multi-selection list box, since it does not allow programmatically reacting to changes in its important characteristic – the position of the current item.

When developing Windows Forms applications, it is preferable to use a list of checkboxes (called a *checked list box*) rather than a multi-selection list box (see the CHECKBOXES project, Chapter 20), since, along with the ability to work with the current (and at the same time, selected) item, the user is able to *mark* any number of list items by setting their checkboxes (thus, when using the checked list box, the program has access to all information about both current and marked list items).

2. To see the change in the current list item during various operations, it is necessary that the list does not lose focus. Therefore, we connected the execution of list operations with *labels* since these controls, unlike usual buttons, cannot receive focus. Thus, there are only two controls on the form that can receive focus: comboBox1 and listBox1. By switching between them using the Tab key, it is easy to check the features of displaying a list with and without focus.

3. In addition to the Beep method without parameters, the Console class has a version of this method with two integer parameters: the first parameter determines the sound *frequency* (in the range from 37 to 32767 Hz), the second determines the sound *duration* (in milliseconds).

19.3. Additional list operations

Place five new labels (label3 – label7) on Form1 and set their properties (since the AutoSize, TextAlign, and BorderStyle properties have the same values for all labels, it is convenient to set them at the same time having previously selected all labels on the form).

Properties

```
label3: Text = Insert, TextAlign = MiddleCenter,
```

```
  AutoSize = False, BorderStyle = FixedSingle
label4: Text = Move Up, TextAlign = MiddleCenter,
  AutoSize = False, BorderStyle = FixedSingle
label5: Text = Move Down, TextAlign = MiddleCenter,
  AutoSize = False, BorderStyle = FixedSingle
label6: Text = Save To File, TextAlign = MiddleCenter,
  AutoSize = False, BorderStyle = FixedSingle
label7: Text = Load From File, TextAlign = MiddleCenter,
  AutoSize = False, BorderStyle = FixedSingle
```

Arrange the new labels as shown in Fig. 19.3.

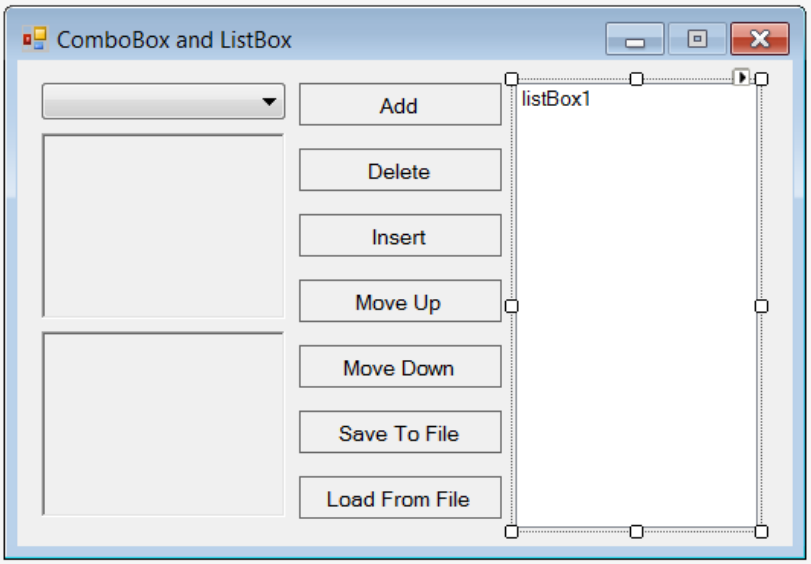


Fig. 19.3. The final view of Form1

At the top of the **Form1.cs** file, add the directive

```
using System.IO;
```

Define the Click event handlers for the added labels:
label3.Click, label4.Click, label5.Click, label6.Click, label7.Click handlers

```
private void label3_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == -1)
    label1_Click(this, null);
  else
  {
    listBox1.Items.Insert(i, comboBox1.Text);
    listBox1.SelectedIndex = i;
  }
}
```

```
private void label4_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i <= 0)
  {
    Console.Beep();
    return;
  }
  object x = listBox1.Items[i];
  listBox1.Items[i] = listBox1.Items[i - 1];
  listBox1.Items[i - 1] = x;
  listBox1.SelectedIndex = i - 1;
}
private void label5_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == -1 || i == listBox1.Items.Count - 1)
  {
    Console.Beep();
    return;
  }
  object x = listBox1.Items[i];
  listBox1.Items[i] = listBox1.Items[i + 1];
  listBox1.Items[i + 1] = x;
  listBox1.SelectedIndex = i + 1;
}
private void label6_Click(object sender, EventArgs e)
{
  if (listBox1.Items.Count == 0)
  {
    Console.Beep();
    return;
  }
  File.WriteAllLines("LISTBOXES.dat",
    listBox1.Items.Cast<string>());
}
private void label7_Click(object sender, EventArgs e)
{
  if (!File.Exists("LISTBOXES.dat"))
  {
```

```
    Console.Beep();
    return;
  }
  listBox1.Items.Clear();
  foreach (var e1 in File.ReadLines("LISTBOXES.dat"))
    listBox1.Items.Add(e1);
  listBox1.SelectedIndex = listBox1.Items.Count - 1;
}
```

Result. Clicking on the **Insert** label inserts a new item in front of the selected one and selects the inserted item (in the case of an empty list, the **Insert** command and the **Add** command perform the same actions). The **Move Up** and **Move Down** commands allow you to move the selected item up and down the list, respectively, while maintaining its selection (when you try to move the first item up or the last item down, a sound signal is generated). The **Save To File** command allows you to save the contents of a non-empty list in the **LISTBOXES.dat** file, the **Load From File** command allows you to load data from this file into the list (if the file is missing, a beep sounds when trying to load data).

Comments

1. When implementing new actions, the Insert (inserting a new item at the specified position) and Clear (clearing the list of items) methods of the Items collection of the ListBox control were used.

2. To check if a file exists, we used the Exists method of the File class.

To write a collection of strings to a text file and read the contents of a text file as a collection of strings, it is convenient to use two static methods of the File class: WriteAllLines and ReadLines. These methods automatically open a file with the specified name, perform the required actions, and then close it.

The WriteAllLines method requires you to specify the collection of strings to write, either as an array or as a sequence (of IEnumerable<string> type); a sequence of the required type can be obtained from the Items collection using the Cast<string> *LINQ query*.

When reading strings from a file, you do not need to perform any typecasting: it is enough to organize a foreach loop, in which all the lines read from the file by the ReadLines method will be processed.

19.4. Performing list operations with the mouse

Connect the DoubleClick event of the listBox1 control to the existing label2_Click handler.

Set the AllowDrop property of the listBox1 control to **True** and add a field iSrc (an *index of source*) to the Form1 class:

```
private int iSrc;
```

This field will contain the index of the item being dragged from listBox1 in drag-and-drop mode:

Define the MouseDown event handlers for the comboBox1 and listBox1 controls and the DragEnter and DragDrop event handlers for the listBox1 control:

comboBox1.MouseDown, listBox1.MouseDown, listBox1.DragEnter, listBox1.DragDrop handlers

```
private void comboBox1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Right)
    DoDragDrop(comboBox1.Text, DragDropEffects.Copy);
}
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Right)
  {
    iSrc = listBox1.IndexFromPoint(e.Location);
    if (iSrc != ListBox.NoMatches)
      DoDragDrop(listBox1.Items[iSrc].ToString(),
        DragDropEffects.Move);
  }
}
private void listBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.All;
}
private void listBox1_DragDrop(object sender, DragEventArgs e)
{
  if (e.AllowedEffect == DragDropEffects.Move)
    listBox1.Items.RemoveAt(iSrc);
  string s = e.Data.GetData(typeof(string)) as string;
  int iTrg = listBox1.IndexFromPoint(listBox1
    .PointToClient(new Point(e.X, e.Y)));
  if (iTrg == ListBox.NoMatches)
    listBox1.SelectedIndex = listBox1.Items.Add(s);
  else
  {
    listBox1.Items.Insert(iTrg, s);
    listBox1.SelectedIndex = iTrg;
  }
}
```

Result. When you double-click on an item in the list, this item is removed (due to connecting the DoubleClick event of the listBox1 control to the label2_Click handler). In addition, you can now use the drag-and-drop mode to move a list item to a new position: just press the *right* mouse button on any (not necessarily selected) list item and drag it to a new location. You can also drag and drop text from the comboBox1 drop-down list into the list box. If the text is dropped onto an existing list item, it is *inserted* at the specified position; if the text is dropped into a free area of the list box, it is *added* to the list. In any case, this item will become selected.

When dragging text from the comboBox1 drop-down list, the cursor image contains the "+" sign, which means the **Copy** drag effect; when dragging the listBox1 list item to a new location, the cursor does not contain the "+" sign, which means the **Move** drag effect. See also Comments 1–3.

Disadvantage. At the beginning of dragging from the drop-down list, the cursor looks like a prohibition sign  .

Correction. Set the AllowDrop property of the comboBox1 control to **True** and define the DragEnter event handler for the comboBox1 control:

comboBox1.DragEnter handler

```
private void comboBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Copy;
}
```

Result. Now, when dragging an item from the drop-down list, the cursor over this list will have a permissive view (although, of course, dropping the item over the drop-down list will not have any effect). Note that, if dragging is started from the *list box*, the cursor above the comboBox1 control will have a prohibition sign; this is quite correct and is due to the fact that the drop-down list as a drop receiver reacts only to the **Copy** drag effect.

Comments

1. We have chosen the *right* mouse button as the initiator of the drag-and-drop mode since both the comboBox1 drop-down list and the listBox1 list box should respond to the left mouse button in the standard way: when left-clicking, the drop-down list expands and the item of the list box is selected. In addition, we have also associated a special action (deleting an item) with double-left-clicking on listBox1. If the drag-and-drop mode were activated by pressing the left mouse button, then the standard actions associated with the left button would be impossible to perform.

2. Methods, events, and properties associated with drag-and-drop mode were described in detail in Chapter 10. The only new feature of this drag-and-drop implementation is the use of the IndexFromPoint(p) method of the ListBox control. This method allows you to determine the index of the list item containing point p with the specified coordinates (if the specified point does not con-

tain a list item, then the method returns the special value ListBox.NoMatches). The IndexFromPoint method is used in the listBox1_MouseDown and listBox1_DragDrop handlers. Since this method requires specifying the *local* coordinates of point p relative to the ListBox control, and only *screen* coordinates can be obtained in the drag-and-drop event handlers, we have to additionally use the listBox1.PointToClient method in the listBox1_DragDrop handler. Note that we do not need to call the listBox1.PointToClient method in the listBox1_MouseDown handler, since the *local* coordinates are returned in the mouse event handlers.

3. Due to the use of different drag effects (**Copy** and **Move**), the user can determine by the type of drag cursor which control is the data source (this is comboBox1 for the **Copy** effect and listBox1 for the **Move** effect). By checking the current drag effect (namely the e.AllowedEffect property) at the beginning of the listBox1_DragDrop method, we determine whether the source item should be removed from the list.

20. Checkboxes and checked list boxes: CHECKBOXES project

The CHECKBOXES project is related to the CheckBox and CheckedListBox controls that implement a checkbox and a list of checkboxes (called a checked list box). Methods for checking and changing the state of checkboxes are considered, as well as the features of using checkboxes that take three states.

20.1. Checkboxes and checking their state

After creating the CHECKBOXES project, place three checkboxes (checkBox1 – checkBox3), three labels (label1 – label3), and a button (button1) on Form1. Set the properties of the form and the added controls and arrange the controls as shown in Fig. 20.1.

Properties (Form1 and its controls)

```
Form1: Text = Checkboxes, MaximizeBox = False,
  FormBorderStyle = FixedSingle, AcceptButton = button1
checkBox1: Text = Group 1, RightToLeft = Yes
checkBox2: Text = Group 2, RightToLeft = Yes
checkBox3: Text = Group 3, RightToLeft = Yes
label1-label3: Text = No items selected
button1: Text = Select Items
```

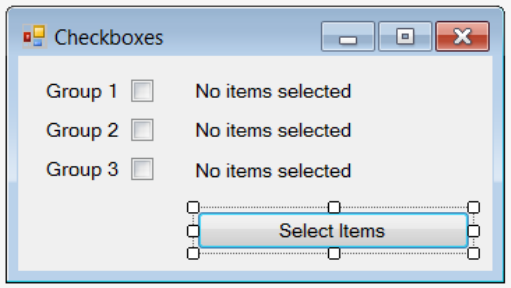


Fig. 20.1. Form1 view

Since the label1 – label3 labels have the same value of the Text property, it is convenient to set this property for all labels at once, after selecting them.

Add a new form to the project (it will be named Form2) and place three CheckedListBox controls on it (they will be named checkedListBox1 – checkedListBox3), as well as the button1 button. Set the properties of Form2 and its controls. When defining the Items property of CheckedListBox controls, a special dialog box is used; in our case, the set of numbers must be input into this dialog box (5 numbers for checkedListBox1, 4 numbers for checkedListBox2, 6 numbers for checkedListBox3), typing each number on a *separate line*.

Properties (Form2 and its controls)

```
Form2: Text = Select Items, MaximizeBox = False,
  MinimizeBox = False, FormBorderStyle = FixedDialog,
  StartPosition = CenterScreen, ShowInTaskbar = False,
  AcceptButton = button1
checkedListBox1: Items = 1 2 3 4 5, CheckOnClick = True
checkedListBox2: Items = 1 2 3 4, CheckOnClick = True
checkedListBox3: Items = 1 2 3 4 5 6, CheckOnClick = True
button1: Text = OK, DialogResult = OK
```

Arrange the controls as shown in Fig. 20.2

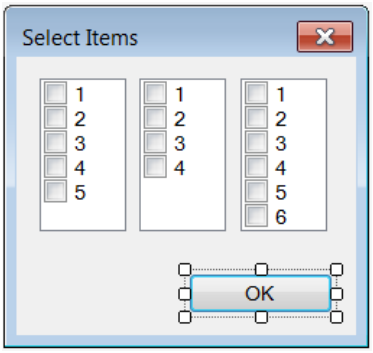


Fig. 20.2. Form2 view

Add a new field to the Form1 class declaration:

```
private Form2 form2 = new Form2();
```

In the constructor of the Form1 class, add the statement

```
AddOwnedForm(form2);
```

Define the Click event handler for button1 on Form1 and the FormClosed event handler for Form2:

button1.Click handler for Form1

```
private void button1_Click(object sender, EventArgs e)
{
  form2.ShowDialog();
}
```

Form2.FormClosed handler

```
private void Form2_FormClosed(object sender,
  FormClosedEventArgs e)
{
  for (int i = 1; i <= 3; i++)
  // processing each group of checkboxes
  {
    string s = "";
    CheckedListBox clb =
```

```
      Controls["checkedListBox" + i] as CheckedListBox;
    int k = clb.CheckedIndices.Count;
    if (k == 0)
      s = "No items selected";
    else
      if (k == clb.Items.Count)
        s = "All items are selected";
      else
      {
        foreach (int j in clb.CheckedIndices)
          s = s + " " + clb.Items[j].ToString();
        s = "Selected:" + s;
      }
    // display information about checked items
    // in the corresponding label of Form1
    Owner.Controls["label" + i].Text = s;
  }
}
```

Result. When you open the **Select Items** dialog box, then check some items in each checkbox group (called a *checked list box*), and finally close the dialog box by clicking **OK** or pressing Enter, the main **Checkboxes** form displays information about the checked items in each group. The checkboxes on the main form are not used yet.

Comments

1. To check an item (or uncheck a checked item) in the checked list box, you can select this item using the arrow keys and press the spacebar. The method of action using the mouse depends on the value of the CheckOnClick property. If this property is set to **False** (the default value), then clicking an item selects this item (makes it highlighted) but *does not change* its state yet: to change the state of the item, you must click on an already selected item again. If the CheckOnClick property is **True**, clicking an item immediately changes its state. The second way seems more natural, since this is how one checkbox (the CheckBox control) behaves by default.

2. You can track user actions related to moving through the list of checkboxes and changing their state using the handlers of two events.

The SelectedIndexChanged event occurs when a user selects an item using the keyboard or mouse, as well as when user clicks by the left mouse button on an already selected item (you can determine the index of the selected item using the SelectedIndex property, which is both readable and writable; indexing, as usual, is from zero). In addition, the SelectedIndexChanged event occurs when the SelectedIndex property is programmatically changed.

The ItemCheck event occurs when a user or program try to change the state of an item; in this case, using the properties of the second parameter e of this event handler, you can determine the previous state (the CurrentValue property), new state (NewValue), and the item's index (Index). The CurrentValue and NewValue properties are of the CheckState enumerated type (with three possible values: Checked, Unchecked, and Indeterminate). It is important to note that the NewValue property can be changed in a handler; this feature is especially useful when using *three-state checkboxes* (see Section 20.3), because user actions *cannot* set a checked list box item in the Indeterminate state (note that the CheckedListBox control, unlike the CheckBox control, does not have the ThreeState property that allows the user to sequentially set a checkbox in each of the three states).

3. The easiest way to get information about checked items is using the CheckedIndices collection property, which contains the indices of the checked items (that is, items in the Checked and Indeterminate state). The number of checked items can be obtained using the Count property of the CheckedIndices collection (of course, the Items collection, which contains all the items, also has the Count property). In addition, there is the CheckedItems collection property, which contains the checked items themselves.

To get the text associated with an item from the CheckedItems or Items collection, apply the ToString method to this item.

If you need to obtain the state of an item, you can use the GetItemCheckState(ind) method of the CheckedListBox control, which returns the state of the item with index ind (the method returns a value of the CheckState enumerated type). The paired method SetItemCheckState(ind, st) allows you to set the item with index ind to the state st. If the Indeterminate state is not used, then it is easier to use the GetItemChecked(ind) method (which returns a boolean value) and the SetItemChecked(ind, val) method with a boolean parameter val. The true value corresponds to a checked item, the false value corresponds to an unchecked item.

20.2. Global setting of CheckedListBox items

Define an event handler for the CheckStateChanged event for checkBox1 and then connect this handler to the CheckStateChanged event of checkBox2 and checkBox3.

checkBox1.CheckStateChanged handler

```
private void checkBox1_CheckStateChanged(object sender,
  EventArgs e)
{
  CheckBox cb = sender as CheckBox;
  string s = "label" + cb.Name[cb.Name.Length - 1];
  Controls[s].Text = cb.Checked ? "All items are selected" :
```

```
    "No items selected";
}
```

Also, change the button1_Click method of Form1:

```
private void button1_Click(object sender, EventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    CheckedListBox clb =
      form2.Controls["checkedListBox" + i] as CheckedListBox;
    bool b = (Controls["checkBox" + i] as CheckBox).Checked;
    for (int j = 0; j < clb.Items.Count; j++)
      clb.SetItemChecked(j, b);
  }
  form2.ShowDialog();
}
```

Result. Checking the checkbox on the main form ensures that all items of the corresponding checked list box will be checked; unchecking the checkbox ensures that all items will be unchecked. When you open the **Select Items** dialog box, its checked list boxes are adjusted.

Disadvantage. Explicitly made settings in the dialog box do not affect the appearance of the checkboxes of the main form. Usually, if *some* of the items are selected in the dialog box, then the corresponding checkbox on the main form is shown in the CheckState.Indeterminate state. The necessary corrections will be made in the next section.

Comment

Instead of the CheckStateChanged event, we could use the CheckedChanged event. We used the CheckStateChanged event because it (unlike the Checked-Changed event) also occurs when the checkbox state changes from **Checked** to **Indeterminate** and vice versa. This feature will be useful in the future.

20.3. Using checkboxes with three states

Modify the Form2_FormClosed method of Form2 as follows:

```
private void Form2_FormClosed(object sender,
  FormClosedEventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    string s = "";
    CheckedListBox clb =
      Controls["checkedListBox" + i] as CheckedListBox;
    CheckBox cb = Owner.Controls["checkBox" + i] as CheckBox;
```

```
    int k = clb.CheckedIndices.Count;
    if (k == 0)
      s = "No items selected";
      cb.Checked = false;
    else
      if (k == clb.Items.Count)
        s = "All items are selected";
        cb.CheckState=CheckState.Checked;
      else
      {
        string s = "";
        foreach (int j in clb.CheckedIndices)
          s = s + " " + clb.Items[j].ToString();
        s = "Selected:" + s;
        Owner.Controls["label" + i].Text = s;
        cb.CheckState = CheckState.Indeterminate;
      }
    Owner.Controls["label" + i].Text = s;
  }
}
```

Result. If *some* of the items are selected in one of the checked list boxes in the **Select Items** window, the corresponding checkbox in the **Checkboxes** window changes its state to Indeterminate. Note that the user cannot explicitly set the checkboxes of the Checkboxes window to the Indeterminate state, because, for the checkBox1 – checkBox3 controls, the ThreeState property is false (the default value).

Remark. Note that, in the new version of the Form2_FormClosed handler, the label text changes only when *some* of the items are checked. If *all* items are checked or *none* of them is checked, then the change in the label text occurs in the checkBox1_CheckStateChanged handler (see Section 20.2), which is automatically called whenever the checkbox state changes.

Error 1. The text **All items are selected** is always displayed near the checkbox, which is in the Indeterminate state. This error is due to the fact that, when the CheckState property is set to Indeterminate, the CheckStateChanged event handler is called, but the Checked property is true in this situation, so the corresponding label obtains the text **All items are selected**.

Correction. Change the checkBox1_CheckStateChanged method of Form1 as follows:

```
private void checkBox1_CheckStateChanged(object sender,
  EventArgs e)
{
```

```
  CheckBox cb = sender as CheckBox;
  if (cb.CheckState == CheckState.Indeterminate)
    return;
  string s = "label" + cb.Name[cb.Name.Length - 1];
  Controls[s].Text = cb.Checked ? "All items are selected" :
    "No items selected";
}
```

Error 2. When you reopen the **Select Items** window, *all* items are checked in the checked list boxes corresponding to the *indeterminate* checkboxes of the main window. This error, like Error 1, is due to the fact that the Checked property is true for both checked and indeterminate checkboxes.

Correction. Change the button1_Click method of Form1 as follows:

```
private void button1_Click(object sender, EventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    CheckedListBox clb =
      form2.Controls["checkedListBox" + i] as CheckedListBox;
    bool b = (Controls["checkBox" + i] as CheckBox).Checked;
    CheckState cs =
      (Controls["checkBox" + i] as  CheckBox).CheckState;
    if (cs == CheckState.Indeterminate)
      continue;
    for (int j = 0; j < clb.Items.Count; j++)
      clb.SetItemChecked(j, b);
      clb.SetItemCheckState(j, cs);
  }
  form2.ShowDialog();
}
```

Result. Now, in the case of indeterminate checkboxes, the corresponding checked list boxes are not changed when the **Select Items** window is opened. For this, the new version of the button1_Click method uses the continue statement, which ensures the immediate termination of the current loop iteration.

21. Viewing images: IMGVIEW project

The IMGVIEW project describes how to use the TreeView and ListBox controls and various classes from the System.IO namespace to visualize the directory structure of a computer. In addition, we describe the SplitContainer and PictureBox controls and discuss issues related to docking and using the Windows registry to store information about the state of the application and then restore it.

21.1. Displaying a directory tree

Place a control of SplitContainer type on Form1 (we will call this control a *split container*). This control will be named splitContainer1 and will immediately occupy the entire client area of the form (since its Dock property is **Fill** by default). The splitContainer1 control consists of two *child panels* (the left one named Panel1 and the right one named Panel2); between these panels there is a *splitter*. Dragging the splitter, you can increase the width of one panel by reducing the width of the other. Each of the child panels can be selected; the properties of the selected panel are displayed in the **Properties** window. The splitter cannot be selected; clicking on it selects the entire splitContainer1 control (note that this is the most convenient way to select a split container, since the rest of it is occupied by child panels).

Remark. There is the Splitter control in the .NET visual control library, but after including the SplitContainer control in .NET 2.0, a separate splitter control is usually not used.

The specifics of working with the SplitContainer control will be discussed in the next section. In this section, we will only use its left panel (Panel1). Place the TreeView control (it will be named treeView1) on this panel and set the properties of the form and this control. Recall that setting the Dock property in the **Properties** window displays a special panel; to select the required **Fill** option in this panel, click on the central rectangle.

Properties

```
Form1: Text = Image Viewer, StartPosition = CenterScreen
treeView1: Dock = Fill, HideSelection = False
```

The form will take the view shown in Fig. 21.1.

Add a new directive to the list of using directives at the beginning of the **Form1.cs** file:

```
using System.IO;
```

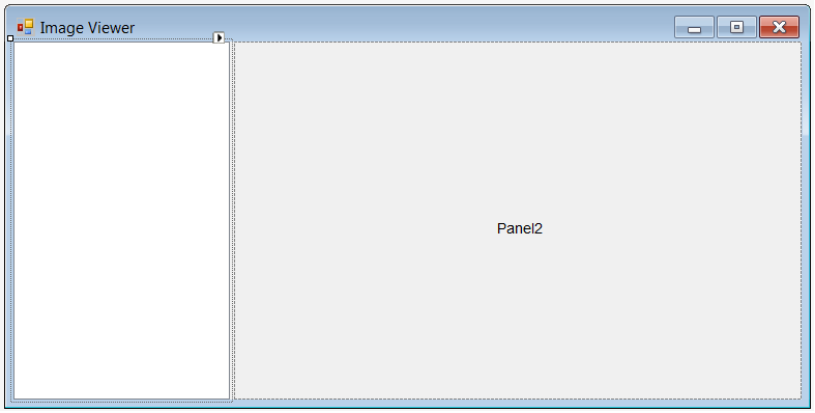


Fig. 21.1. Form1 view at the initial stage of development

In the Form1 class, define a helper method:

```
void MakeChildNodes(TreeNode node)
{
  node.Nodes.Clear();
  if (node.Tag.ToString() == "")
    foreach (var drive in DriveInfo.GetDrives())
    {
      if (!drive.IsReady)
        continue;
      TreeNode newNode = new TreeNode();
      newNode.Tag = drive.RootDirectory.FullName;
      newNode.Text = drive.Name;
      if (drive.VolumeLabel != "")
        newNode.Text += "[" + drive.VolumeLabel + "]";
      if (drive.RootDirectory.GetDirectories().Length > 0)
        newNode.Nodes.Add("*");
      node.Nodes.Add(newNode);
    }
  else
  {
    try
    {
      foreach (var subDir in Directory
        .GetDirectories(node.Tag.ToString()))
      {
        try
        {
          TreeNode newNode = new TreeNode();
          newNode.Tag = subDir;
```

```
          newNode.Text = Path.GetFileName(subDir);
          if (Directory.GetDirectories(subDir).Length > 0)
            newNode.Nodes.Add("*");
          node.Nodes.Add(newNode);
        }
        catch {}
      }
    }
    catch {}
  }
}
```

Define an event handler for the BeforeExpand event for the treeView1 control:

treeView1.BeforeExpand handler

```
private void treeView1_BeforeExpand (object sender,
  TreeViewCancelEventArgs e)
{
  MakeChildNodes(e.Node);
}
```

Add the following statements to the constructor of the Form1 class:

```
TreeNode node = new TreeNode();
node.Tag = "";
node.Text = "My Computer";
node.Nodes.Add("*");
treeView1.Nodes.Add(node);
treeView1.SelectedNode = node;
```

Result. When you start the program, the left panel of the form displays the **My Computer** *root node* of the *tree* (*hierarchical list*) of the directory structure of computer. When you *expand* this node, all currently available disks appear; when you expand any disk, all its first-level directories appear. If the disk has a label, it is displayed in square brackets next to the disk letter.

If the directory contains subdirectories, then it can also be expanded. Near those nodes that can be expanded, a special *expand marker* is displayed, usually in the form of a square with a "+" sign (Fig. 21.2). When expanding a node, the marker view changes to a square with a "–" sign. To expand/collapse a node, just click on the expand marker or double-click on the name of the required node. A single click on the node name selects it.

You can also navigate the tree using the keyboard; the Up and Down arrow keys move the selection to a node located above or below the current one, and the Right and Left arrow keys expand the node (if it can be expanded) or, respectively, collapse the node (if it is expanded). If the node is already collapsed

or there is no expand marker associated with it, then the Left key navigates to the previous level directory (that is, the *parent* node for this node).

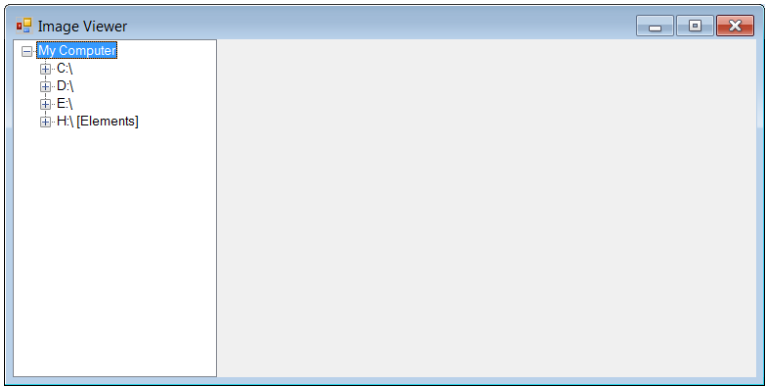


Fig. 21.2. View of the running IMGVIEW application at the initial stage of development

Comments

1. The main feature of the TreeView control, which allows creating trees (hierarchical lists), is that each of its nodes (of TreeNode type) itself can be a root of a tree. Checkboxes can be associated with tree nodes; it is enough to set the CheckBoxes property of the TreeView control to **True**. In this case, the tree will contain not only the *selected* item, but also a set of *checked* items. To check or uncheck a checkbox, you can either click on the checkbox or press the space bar.

The main events of the TreeView control are BeforeExpand, BeforeCollapse, BeforeSelect, and BeforeCheck; these events occur when the user performs the actions to expand, collapse, select, or check/uncheck some node of the tree, respectively. You can determine for which tree node an event occurred using the Node property of the parameter e of the handler. In addition, the parameter e has the Cancel property that allows you to cancel the specified action (for this, you just need to set e.Cancel equal to true). There are also the AfterExpand, AfterCollapse, AfterSelect, and AfterCheck events of the TreeView control.

In addition, the TreeView control has the SelectedNode property that allows you to determine or change the selected item. There is no SelectedIndex property for a tree.

Both the TreeView and TreeNode classes have the Nodes collection property that contains all the *first-level* children of the tree or node. In addition, the TreeNode class has the follows read-only properties: Parent (returns the parent node of a given node), Index (the index of the node in the Nodes collection of its parent), and TreeNode (a reference to the tree that this node belongs to). The name of the node that appears in the tree is contained in the Text property, which is available for reading and writing.

2. The program uses classes from the System.IO namespace: the DriveInfo class, which allows you to get information about disk devices of the computer (this class was previously used in the DISKINFO project, see Chapter 2), and

the DirectoryInfo and Directory classes, which allow to get information about the directory.

For the DriveInfo class, we use the static GetDrives method, which returns a collection of all available disk devices (each member of the collection is also of DriveInfo type), and the IsReady, Name, VolumeLabel, and RootDirectory properties. The IsReady property checks the availability of the device, Name returns its name, and VolumeLabel returns the name of the label (the last two properties are used to form a node name corresponding to the root directories of the available devices). The RootDirectory property is of DirectoryInfo type. Using the DirectoryInfo object, it is easy to determine all the characteristics of the directory: its name (Name), full name (FullName), an array of DirectoryInfo[] type with information about all its *subdirectories* (obtained using the GetDirectories method), an array of FileInfo[] type with information about all its *files* (obtained using the GetFiles method), etc.

The Directory class also has GetDirectories and GetFiles methods. These methods are static; they are useful when you need to get a set of subdirectories or files for a directory with a *known name*, which is specified as a method parameter. In the second part of the MakeChildNodes method, we use the Directory.GetDirectories method, as well as the GetFileName static method of the Path class from the same System.IO namespace. This method extracts the name, without the preceding path, from the fully qualified filename or directory.

3. When using a directory tree in a program, you should not build the entire tree at once, since the number of directories on hard drives is usually very large. A more efficient approach is to build a part of the tree when it needs to be displayed on the screen. An additional advantage of such an approach is that it allows you to easily *update* any part of the tree in a situation when you need to take into account changes in the set of its nodes: to do this, you just collapse this part and re-expand it.

To simplify the steps for displaying new devices, we create a tree with the **My Computer** root node which contains all currently available devices. If a new device is connected to the computer (for example, a USB drive) or if an existing device becomes available (for example, as a result of loading a CD or DVD into a device for reading them), then, to display these new devices, you should simply collapse and re-expand the **MyComputer** root node.

We use a simple way to mark a tree node as expandable: a “dummy” child node with name “*” is added to the Nodes collection of that node. This child is never displayed on screen because we remove it and build a set of “real” child nodes before expanding the parent node. Some authors suggest adding such a dummy node to *any* tree node that you create. In this case, when you try to expand a node that has no children, its expand marker simply disappears. We did it differently: in our program, a dummy child is added only when a node *actually* has children. This method requires additional time for finding children

(which we do using the GetDirectories method), but is more user-friendly, as it immediately shows the presence of child nodes. Note that, by default, the GetDirectories method only returns *first-level* subdirectories.

Also note the use of the Tag property of the TreeNode element to store the *full path* to the directory that this tree node is associated with. For disk drives, the full path to their *root directory* is stored. There is only one node in the directory tree that is not associated with a directory: the **My Computer** root node whose the Tag property is an empty string.

4. The ExpandItem method uses two try – catch statements to handle errors related to accessing directories. The outer try block is designed to correctly handle a situation in which an attempt is made to access a directory that has become inaccessible (either due to its deletion, or due to the disconnection of the device containing this directory). In such a situation, expanding is canceled and the expand marker disappears near this node. The inner try block (located in the foreach loop) is intended to correctly handle the situation in which some child directories block access to their contents (in this case, calling the GetDirectories method for those directories will result to the exception). Such directories are simply not included in the list of child directories.

5. We changed the value of the HideSelection property to **False** so that the selected node in the TreeView control is highlighted even when the control loses focus (this property was previously described in Comment 2 of Section 8.1).

Disadvantage. Each time the program is launched, the user has to perform the same actions to go to the desired directory.

Correction. Add a new helper method to the Form1 class:

```
TreeNode InitialExpanding(string fullPath)
{
  if (!Directory.Exists(fullPath))
    return null;
  string[] paths = fullPath.Split('\\');
  paths[0] += "\\";
  TreeNode rootNode = treeView1.Nodes[0];
  MakeChildNodes(rootNode);
  rootNode.Expand();
  TreeNode node = rootNode;
  foreach (var e in paths)
  {
    node = node.Nodes.Cast<TreeNode>()
      .First(e1 => e1.Text.ToUpper() == e.ToUpper());
    MakeChildNodes(node);
    node.Expand();
  }
```

```
  return node;
}
```

Define the Load event handler for the Form1 class:

Form1.Load handler

```
private void Form1_Load(object sender, EventArgs e)
{
  var node = InitialExpanding(Directory.GetCurrentDirectory());
  if (node != null)
    treeView1.SelectedNode = node;
}
```

Result. Now, when the program starts, it automatically switches to the current directory (by default, it is the **bin\Debug** subdirectory, which is included in the directory containing the developing project, see Fig. 21.3).

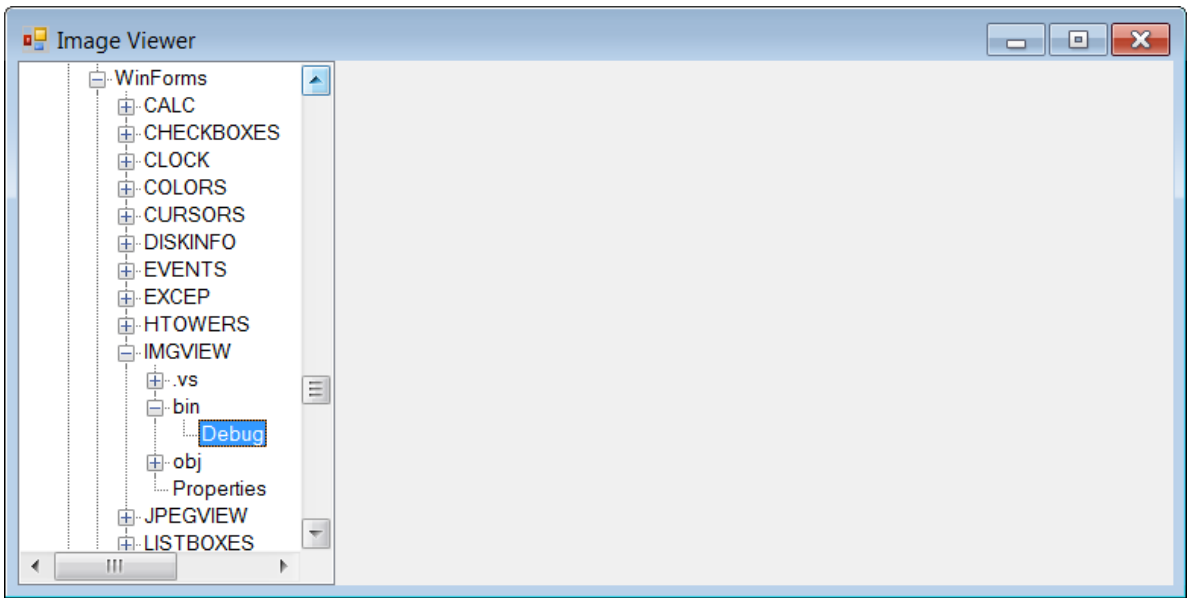


Fig. 21.3. View of the running IMGVIEW application with the selected current directory

Comments

1. In the future (see Section 21.6), we will modify the project in such a way that, when the program starts, it switches to the directory that was selected before previous closing of the program. For this reason, the InitialExpanding method checks existing the required directory. If the directory is missing, the method returns null. To check for the existence of a directory with the required name, we use the static Exists method of the Directory class. The static method of the File class of the same name allows you to check the existence of a file.

2. To split the full path to the initial directory into parts, a version of the Split method of the string class with a character parameter defining the separator (in our case, the '\' character) is used. Since the slash is required in the root directory name, it is added to the first element of the generated array.

3. The InitialExpanding method first expands the root node of the treeView1 control (this is the only child node in its Nodes collection) and then iterates over all the directories included in the full path to the initial directory. The next subdirectory included in the path of the initial directory is searched for in the corresponding collection of child nodes of the current node. After finding it, it expands resulting in a new set of nodes in the tree, in which the next part of the path to the initial directory is searched. The search uses the First *LINQ query*, which returns the first item in the collection that satisfies the specified condition. Because directory names are case insensitive, they are converted to uppercase by the ToUpper method before comparing them.

4. The Expand method of the TreeNode class is intended for programmatically expanding a tree node. When this method is called, the BeforeExpand event does not occur, so you must first explicitly call the MakeChildNodes helper method, which creates a set of child nodes for this node.

4. Notice that the InitialExpanding method expands the items in the tree, but does not select them. This is due to the fact that the event of selecting a tree node will be associated later (see Section 21.2) with the execution of additional actions that can run for a long time. Therefore, the selection of a new node is performed after exiting the InitialExpanding method (if the method call was successful and did not return null).

21.2. View images from image files in the selected directory

Place a new split container on the Panel2 panel of the splitContainer1 control (the added split container will be named splitContainer2). For each split container, set the TabStop property to **False**.

Although the second split container is placed on one of the panels of the first split container, visually, the form will contain three panels of the same height and with splitters of the same width. You can distinguish panels by using the upper part of the **Properties** window. Sequential selection of form panels from left to right will indicate the following names at the top of the **Properties** window: splitContainer1.Panel1, splitContainer2.Panel1, splitContainer2.Panel2. Note that clicking on the splitter located between the two *left* panels will select the splitContainer1 control and clicking on the splitter between the two *right* panels will select the splitContainer2 control.

Place the ListBox control on the Panel1 panel of the splitContainer2 control and the PictureBox control on the Panel2 panel of the splitContainer2 control (the added controls will be named listBox1 and pictureBox1). Set the properties of the added controls (see also Fig. 21.4).

Properties

```
listBox1: Dock = Fill, IntegralHeight = False
pictureBox1: Dock = Fill, SizeMode = Zoom, BorderStyle = Fixed3D
```

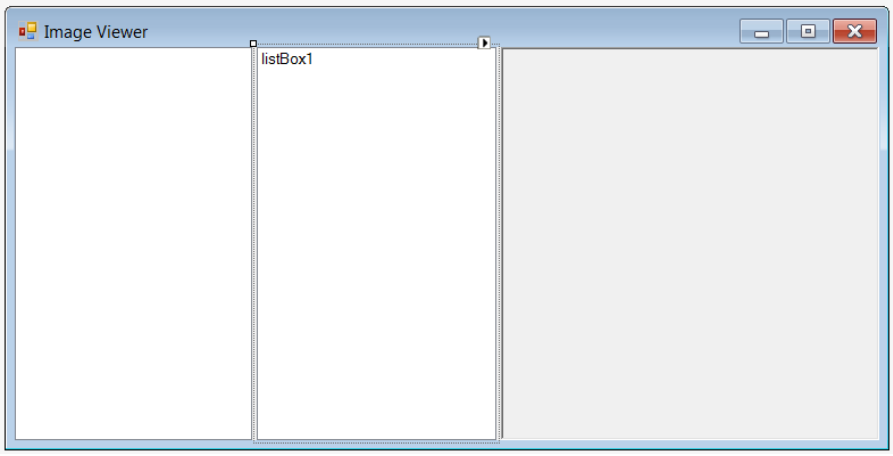


Figure: 21.4. Form1 view at the second stage of development

Add the declaration of the imageExts field to the beginning of the Form1 class declaration:

```
string [] imageExts = {".bmp", ".jpeg", ".jpg", ".png", ".gif",
  ".ico", ".wmf", ".emf"};
```

The imageExts array contains various image file extensions.

Define handlers for the AfterSelect event for the treeView1 control and the SelectedIndexChanged event for the listBox1 control:

treeView1.AfterSelect and listBox1.SelectedIndexChanged handlers

```
private void treeView1_AfterSelect(object sender,
  TreeViewEventArgs e)
{
  listBox1.Items.Clear();
  string path = e.Node.Tag.ToString();
  if (!Directory.Exists(path))
    return;
  foreach (string file in Directory.GetFiles(path)
      .Select(e1 => Path.GetFileName(e1))
      .Where(e1 => imageExts.Contains(Path.GetExtension(e1)
         .ToLower()))
      .OrderBy(e1 => e1))
    listBox1.Items.Add(file);
}
private void listBox1_SelectedIndexChanged (object sender,
  EventArgs e)
{
  string name = listBox1.SelectedItem.ToString();
  Text = "Image Viewer -" + name;
  UseWaitCursor = true;
  try
```

```
  {
    pictureBox1.Image =
      new Bitmap(treeView1.SelectedNode.Tag + "\\" + name);
    Text += "(" + pictureBox1.Image.Width + "x"
      + pictureBox1.Image.Height + ")";
  }
  catch
  {
    pictureBox1.Image = null;
    Text += "(WRONG FORMAT)";
  }
  UseWaitCursor = false;
}
```

Result. When you select a directory in the directory tree, the file list displays the names of image files (with the .**bmp**, .**jpeg**, .**jpg**, .**png**, .**gif**, .**ico**, .**wmf**, .**emf** extensions) located in the selected directory. File names are sorted alphabetically. See also Comment 1.

When you click on the file name (or when moving through the file list using the arrow keys) the pictureBox1 control in the right-hand panel displays an image from the selected file. The image is scaled to fit the panel while maintaining aspect ratio, and the full file name and image size (in pixels) are indicated in the form title bar (see Fig. 21.5).

If the current directory does not contain image files or if the current image file has an incorrect format, the viewing area remains empty and, in case of an incorrect format, the error information (the **(WRONG FORMAT)** text) is displayed in the form title bar. Note that, for testing purposes, an empty file was added to the **Debug** directory (see Fig. 21.6).

When an image is loading, the cursor appearance for the entire application changes to a wait cursor; after loading is complete, the cursor appearance is restored (working with cursors was discussed in detail earlier in the CURSORS project, see Chapter 11). This feature is useful when loading very large images.

The width of the panels can be changed by dragging the splitters between them (when you move the mouse cursor over these dividers, it becomes a double-headed arrow). When you change the width of the form, the width of each of the three panels changes proportionally. Pressing the Tab key allows you to toggle between the directory tree and the file list. See also Comments 2–4.

Remark. Since, at the launching of the program, the selected directory is the directory from which the exe-file is run (i. e., the **bin\Debug** subdirectory of the directory with the IMGVIEW project), to speed up testing of the IMGVIEW project, it is advisable to copy several graphic files of different types into the **bin\Debug** subdirectory of the project directory.

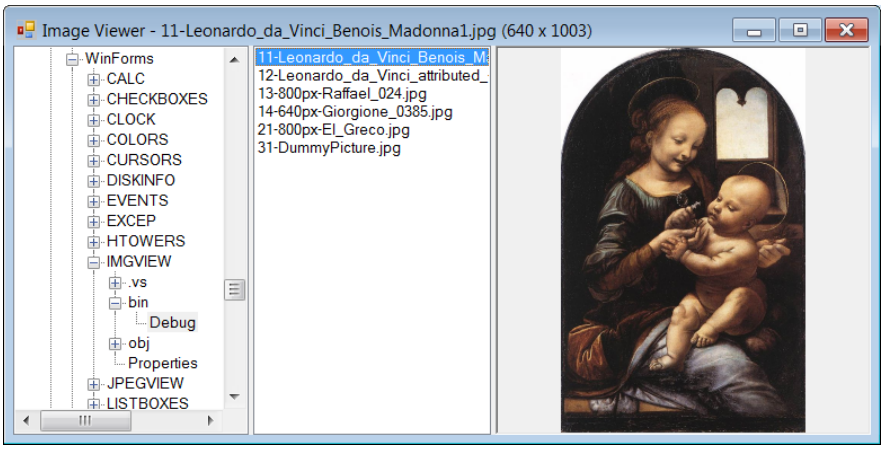


Fig. 21.5. View of the running IMGVIEW application with the loaded image

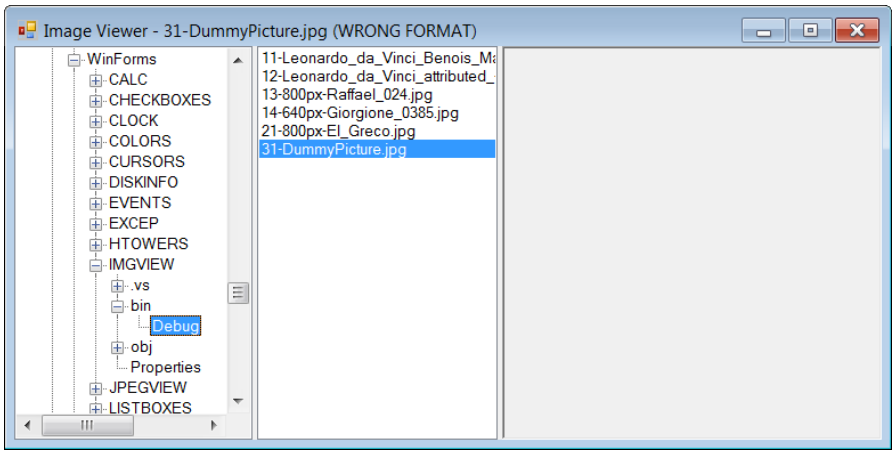


Fig. 21.6. View of the IMGVIEW application when a file in wrong format is selected

Disadvantage 1. When the form is resized (and the width of its panels is changed synchronously), the contents of the directory tree and the file list flicker as a result of repeated redrawing.

Correction. For the splitContainer1 and splitContainer2 controls, set their FixedPanel property to **Panel1**.

Result. Now changing the width of the form affects only the right-hand panel containing the image; the width of panels with directory tree and file list does not change. In such a situation, the flickering of the directory tree and file list practically disappears. See also Comment 5.

Disadvantage 2. When changing the directory, the previously loaded image remains in the right-hand panel. In addition, at the beginning of the program, none of the images from the selected directory is displayed.

Correction. Add a statement to the beginning of the treeView1_AfterSelect method

```
pictureBox1.Image = null;
Text = "Image Viewer";
```

and add a conditional statement at the end of the same method:

```
if (listBox1.Items.Count > 0)
```

```
listBox1.SelectedIndex = 0;
```

Result. Now, when changing the directory, the first item is automatically selected in the file list and its image is displayed in the right-hand panel. If the directory does not contain image files, the right-hand panel is cleared. See also Comment 6.

Comments

1. To get a list of all files from a given directory, we use the GetFiles method of the Directory class. Then we successively apply three LINQ queries to the resulting set of file names: the Select query returns a set of file names without the previous path, the Where query selects image file names with the required extensions, the OrderBy query orders the names alphabetically. All obtained file names are added to the Items collection of the listBox1 control.

2. By nesting split containers into each other, it is possible to provide an arbitrarily complex configuration of panels with the possibility of flexible adjustment of their sizes. The problem when working with nested split containers is their selection (for example, to set properties in the **Properties** window). If the child panel is not selected, then you can select the split container by clicking on its splitter. If the child panel or the controls located on it are selected, then it is easier to press the Esc key several times, sequentially going to the parent controls of all levels (up to the form). Let us give an example related to the current project. If we assume that the pictureBox1 control is currently selected, then, by pressing the Esc key several times, we will sequentially select the following controls:

- splitContainer2.Panel2;
- splitContainer2;
- splitContainer1.Panel2;
- splitContainer1;
- Form1.

In the case of a complex hierarchy of controls, the **Document Outline** window is useful, which is displayed on the screen using the **View | Other Windows | Document Outline** menu command. See Fig. 21.7 with the view of this window for our project at the current stage of its development.

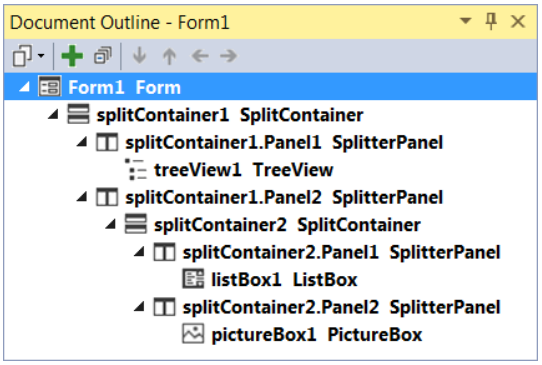


Fig. 21.7.The Document Outline window

3. We set the IntegralHeight property to **False** for the listBox1 control. This property determines whether to display the top part of the last on-screen list item if the entire item cannot be displayed on the screen. When set to **True** (the default value for the ListBox control), the list displays only those items that are *fully visible* on the screen. In our case, this will lead to the situation when the listBox1 control will not be aligned to the bottom border of the form (an empty space will be displayed between the bottom border of the listBox1 control and the bottom border of the form). To prevent this, you just need to set the IntegralHeight property to **False**.

4. The try block is used in the listBox1_SelectedIndexChanged handler in order to correctly handle a possible situation when a file with an extension corresponding to image files has an incorrect format and therefore cannot be displayed in the PictureBox control. In this situation, calling the constructor of the Bitmap class throws an exception, which is immediately handled in the catch section. The exception handling is that the PictureBox control becomes empty, that is, does not contain an image.

5. The FixedPanel property is one of the properties of the split container that allows flexible customization of the behavior of the panels associated with it. Note also the Orientation property, which determines the orientation of the splitter and, accordingly, the location of the child panels of the split container. The possible values of the Orientation property are **Vertical** (vertical splitter, Panel1 on the left, Panel2 on the right) and **Horizontal** (horizontal splitter, Panel1 on top, Panel2 on the bottom). When describing other properties, we will use the concept of the *thickness* of the split container and panel. When the split container is oriented vertically, we mean the *width* of the control by its thickness, and when it is oriented horizontally, we mean its *height* by its thickness.

Let us go back to the FixedPanel property. If it is equal to **None** (the default value), then, when the container thickness changes, the thickness of its panels changes *proportionally*. If the FixedPanel property is **Panel1** or **Panel2**, then the thickness of the specified panel does not change when the thickness of the container changes.

The Panel1MinSize and Panel2MinSize properties (default 25) define the minimum thickness of the corresponding panel. Using the two paired properties Panel1Collapsed and Panel2Collapsed, you can hide and restore the corresponding panel by setting the property to **True** or **False** (the remaining panel will take up the entire area of the container, even if it was previously hidden; therefore, you cannot hide both panels at once).

The position, width, and step of changing the splitter are determined by the SplitterDistance, SplitterWidth and SplitterIncrement properties, respectively. By default, the width is 4 pixels and the step is 1 pixel. You can block the ability to change the position of the splitter by the user by setting the IsSplitterFixed prop-

erty to **True** (while maintaining the ability to change the position of the splitter programmatically).

It should be noted that the position of the splitter can be changed not only by dragging it with the mouse, but also with the arrow keys, if you first move the focus to the splitter using the Tab key. However, such a possibility is not always useful. For example, if there are two panels with a splitter on a form, the user usually expects the Tab key to allow him to switch between panels. Therefore, in most programs, it is reasonably to disable focus on the splitter. For this, it is enough to set the TabStop property of the split container to **False**.

To track user actions related to changing the position of the splitter, the split container has two events: SplitterMoving, which occurs when an attempt is made to change the position of the splitter, and SplitterMoved, which occurs when its position is successfully changed. The SplitterMoving event, like all events raised *before* the execution of an action, allows it to be canceled by setting the e.Cancel value to true in the event handler (e is the second parameter of the handler).

6. Of the two events associated with a change in the selection of a node in the TreeView control (BeforeSelect and AfterSelect), we used the AfterSelect event, since a new node has already been selected in the tree when it occurs. When the BeforeSelect event occurs, the new node has not yet been selected, and although it can be determined in the event handler using the e.Node property, this information is not available to the listBox1_SelectedIndexChanged event handler.

21.3. Docking of controls and its features

In this section, we will place the ComboBox control at the bottom of panel 1 of the splitContainer2 control. When adding this control to the form, it is enough to place it in the listBox1 control (which completely overlaps the panel 1 of the splitContainer1 control) and then set the following properties for the added comboBox1 control:

Properties

```
comboBox1: Dock = Bottom, DropDownStyle = DropDownList
```

Error. The new control is placed at the lower part of the panel 1 (that is, it is "docked" to its bottom border), however, it is located *above* the listBox1 control overlapping its lower part (you can see this if you display a *large* set of image files; as a result, a vertical scroll bar appears in the file list, and its lower part *will be hidden* by the comboBox1 control).

Correction. Select the comboBox1 control and click on the **Send to Back** button (this button is located on the right side of the **Layout** panel).

Result. Now the listBox1 control (with the Dock property equal **Fill**) occupies the entire area of panel 1 of the split container, *except* for the lower part, which is reserved for the comboBox1 control (see Fig. 21.8). We will add the necessary functionality for the comboBox1 control in the next section.

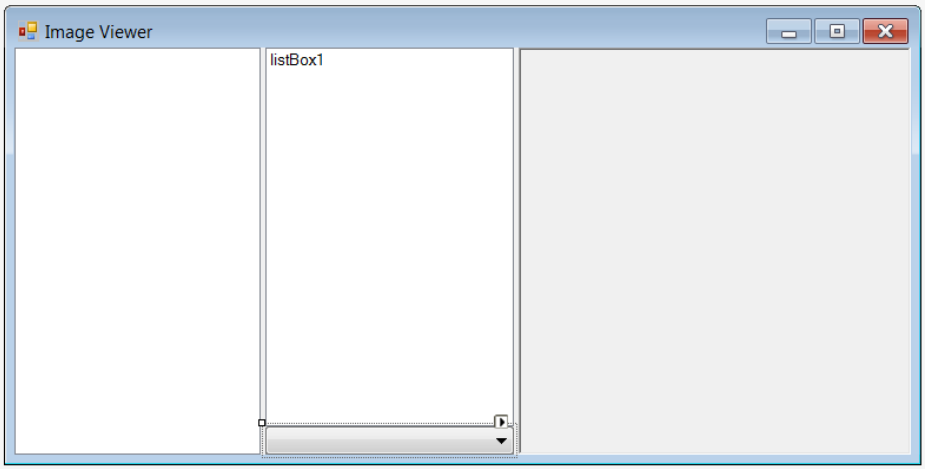


Fig. 21.8. The final view of Form1

Comment

Using the Dock property, which is available for all visual controls, you can easily arrange the relative positioning of controls in a form so that they occupy the entire client area of the form and do not overlap each other even when the form is resized. To do this, all controls, except one, must be docked to one of the form borders (the Dock properties of these controls must have the values **Top**, **Bottom**, **Left**, or **Right** depending on which form border they should be docked to), and the remaining control must *fill* the area of the form not occupied by the controls docked to the borders (the Dock property of this control must be equal to **Fill**).

It is necessary to discuss the following question: if two controls are docked to the same border, which one will be closer to this border? When using a form designer, you can give the following answer (though, as we will see later, it will not be entirely correct): *the control placed earlier on the form will be closer to the border*. This rule corresponds to the developer's intuition: indeed, the previously added control "captures" the part of the form near the border and will not "give" it to another control that appears later. It is important to emphasize that the order in which values are assigned to the Dock properties (using the **Properties** window) *does not matter*: if, for example, panel1 was placed on the form earlier than panel2, then even if the **Bottom** value was first set for the panel2.Dock property and *then* for the panel1.Dock property, panel1 will still be located closer to the bottom border.

So, everything is determined by the *order of placing the controls in the form*. But this order is closely related to the *z-order* of controls (in other words, to the order of their placement in the Controls collection of the parent control). We discussed the *z*-order features in detail in Section 9.1, Comment 4. In particular, we established that, if the panel1 control was placed on the form in design mode *before* panel2, then its *z*-coordinate (as well as the index in the Controls collection) will have a *greater* value (for example, if the form contains on-

ly two of these controls, then panel1 will have index 1 and panel2 will have index 0). But this, in turn, means that the panel2 control will be the *first* item of the Controls collection. Recall that, to add a control to the Controls collection, it is enough to use its Add method. Thus, we come to another rule, which is no longer quite intuitive: a control that was *later* added to the Controls collection of the parent control (and, accordingly, has a greater index in it) will be *closer* to the border of the parent control.

Now it should become clear the way we corrected the error noted in this section: in order for the comboBox1 control to "capture" the bottom of the panel, we had to send it to the *end* of the *z*-sequence by clicking the **Send to Back** button (thereby, we set the *greatest index* for it in the Controls collection).

In conclusion, we make two remarks.

1) As we have seen, even if the controls are added to the form in an order that does not correspond to the order of their docking, we can always correct the position by changing their *z*-order using the **Send to Back** and **Bring to Front** commands.

2) If visual controls are created and added to the form not in design mode, but by programmatically, then, for the correct order of docking, you must follow the extremely "unintuitive", although the simple rule "*work from the center*": first you need to add a control with the Dock property equal to **Fill**, then the controls around it, and finally the controls that should be docked directly to the border of the form.

21.4. Setting the image view mode

Add the following statements to the constructor of the Form1 class:

```
comboBox1.Items.Add(PictureBoxSizeMode.Zoom);
comboBox1.Items.Add(PictureBoxSizeMode.StretchImage);
comboBox1.Items.Add(PictureBoxSizeMode.Normal);
comboBox1.SelectedIndex = 0;
```

Define an event handler for the SelectedIndexChanged event for the comboBox1 control:

comboBox1.SelectedIndexChanged handler

```
private void comboBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  pictureBox1.SizeMode =
    (PictureBoxSizeMode)comboBox1.SelectedItem;
}
```

Result. Using the comboBox1 drop-down list, you can select three *image view modes*: in the **Zoom** mode, the image is scaled while maintaining aspect ratio as before; in **StretchImage** mode, the image is stretched to the entire right-hand panel; in **Normal** mode, the image is displayed without scaling.

Disadvantage. When displaying large images without scaling, it is advisable to add scroll bars to view different parts of the image.

Correction. In the **Properties** window for the Panel2 object of the SplitContainer2 control, set the AutoScroll property to **True**. In the constructor of the Form1 class, change the statement

```
comboBox1.Items.Add(PictureBoxSizeMode.Normal);
```

in the following way:

```
comboBox1.Items.Add(PictureBoxSizeMode.AutoSize);
```

Add new statements to the comboBox1_SelectedIndexChanged method:

```
if (pictureBox1.SizeMode == PictureBoxSizeMode.AutoSize)
  pictureBox1.Dock = DockStyle.None;
else
  pictureBox1.Dock = DockStyle.Fill;
```

Result. The third mode is now called **AutoSize**; in this mode, scroll bars appear on the panel when large images are displayed. Fig. 21.9 shows the IMGVIEW application window in different image view modes.

Comments

1. When defining a set of items for the comboBox1 drop-down list, we used the objects of the PictureBoxSizeMode enumeration, which defines various scaling options for the PictureBox control. Due to this, we were able to directly assign the value of the current list item to the SizeMode property in the comboBox1_SelectedIndexChanged method (it was only required to perform the appropriate type conversion). Note that string representations of PictureBoxSizeMode type are used as the text of the drop-down list items.

2. The SizeMode property is responsible for setting the relative position of the PictureBox control and the image loaded into it, as well as for image scaling. Its value options are **Zoom**, **StretchImage**, **Normal**, and **AutoSize**. There is also a **CenterImage** option, which differs from the **Normal** option only in that the image is placed in the center of the PictureBox control (whereas, in the **Normal** mode, it is in the upper left corner of the control). In the **Normal** (and **CenterImage**) mode, the picture is cropped if it exceeds the size of the control. To correct this, we changed the **Normal** mode to the **AutoSize** mode, which resizes the *control* to fit the loaded image (in this only case, the *control* is resized to fit the image, in other scaling modes the *image* is resized to fit the control).

To make the pictureBox1 control resizable in the case of the **AutoSize** mode, we disable the **Full** docking mode for the pictureBox1 control. In addition, we set the AutoScroll property of the panel containing the **PictureBox** control to **True** (the default value is **False**). If AutoScroll is **True**, scroll bars appear on the panel if it contains controls that are larger than the panel itself. Note that the AutoScroll property is avaliable not only for split container panels, but also for usual panels (the Panel control), as well as for the form.

Fig. 21.9. View of the IMGVIEW application in different image view modes

21.5. Saving information about the state of the program in the Windows registry

At the beginning of the **Form1.cs** file, add the following directive:

```
using Microsoft.Win32;
```

Add a declaration of the new field to the Form1 class declaration:

```
string regKeyName = "Software\\WinFormsExamples\\IMGVIEW";
```

Define an event handler for the FormClosed event for Form1:

Form1.FormClosed handler

```
private void Form1_FormClosed(object sender,
  FormClosedEventArgs e)
{
  RegistryKey rk = null;
  try
  {
    rk = Registry.CurrentUser.CreateSubKey(regKeyName);
    if (rk == null)
      return;
    rk.SetValue("FormWidth", Width);
    rk.SetValue("FormHeight", Height);
    rk.SetValue("Split1", splitContainer1.SplitterDistance);
    rk.SetValue("Split2", splitContainer2.SplitterDistance);
    rk.SetValue("Zoom", comboBox1.SelectedIndex);
    rk.SetValue("Path", treeView1.SelectedNode.Tag.ToString());
    rk.SetValue("File", listBox1.SelectedIndex);
  }
  finally
  {
    if (rk != null)
      rk.Close ();
  }
}
```

Result. Now, when the program exits, information about its current state is written to the *Windows registry*. In order to verify this, you should run the **Registry Editor** program. The easiest way to launch this program is by executing the **Run...** command from the Windows **Start** menu and entering the text **regedit** in the window that appears. In the registry editor, select the **HKEY_CURRENT_USER** *registry key* (that is, a root section of the registry) and the **Software\WinFormsExamples\IMGVIEW** *registry subkey* (that is, a subsection) in it. As a result, the *regstry values* of the selected subkey will be displayed in the right panel of the registry editor (each registry value has name

and data). The subkey should contain seven values (in addition to the **Default** value, which is not used in our program) with the names **File**, **FormHeight**, **FormWidth**, **Path**, **Split1**, **Split2**, **Zoom**. The data of the **Path** value is of string type, the other data are integers. In Section 21.6, we will add a fragment to the program that allows to read this data from the registry.

Comments

1. The Windows registry is a convenient centralized storage of data necessary for the correct working of programs, in particular, to restore their settings at the next launch. If for each computer user it is desirable to store his own settings, then they should be placed in the **HKEY_CURRENT_USER** registry root key. If the settings are to be the same for all users, then they should be placed in the **HKEY_LOCAL_MACHINE** registry root key. In any case, in the selected root key, you must create a subkey associated with a specific program (usually this subkey is placed in the **Software** subkey of the selected root key). The .NET library provides the Registry and RegistryKey classes defined in the Microsoft.Win32 namespace to access registry data.

The Registry class lets you select one of the root registry keys and get an associated object of RegistryKey type. To retrieve standard root keys, there are static read-only properties of the Registry class. The CurrentUser property corresponds to the **HKEY_CURRENT_USER** root key, the LocalMachine property corresponds to the **HKEY_LOCAL_MACHINE** root key.

Once you obtain an object of RegistryKey type using the Registry class, you can use it to create new subkeys and open existing subkeys in read and/or write modes. The CreateSubKey method opens a subkey in write mode; if this subkey does not exist, then the method creates it. The OpenSubKey method opens an existing subkey in read-only mode. In both methods, the full path to the required subkey must be specified as a string parameter. There is also an overloaded version of the OpenSubKey method that allows you to open an existing subkey in read/write mode; to do this, you must specify the second, additional parameter equal to true in the method. If the requested operation is successful, both methods return an object of RegistryKey type associated with the open subkey. If the operation fails, then either a null value is returned (for example, if an attempt is made to open a missing subkey using the OpenSubKey method), or an exception is raised (for example, if the program does not have sufficient rights to access the specified subkey in the required mode). Any successfully opened subkey must be closed using the Close method.

The SetValue method of the RegistryKey class allows you to add or change values for an open subkey. It has two parameters: the name of the value and data associated with it (data can be of string type, integer type, or byte array type). The method for obtaining data from the subkey values will be described in Section 21.6.

2. To debug fragments of the program related to the registry, you must use the **Registry Editor** program, since it allows you to view and edit the registry contents. If you find out that a subkey or any of its values were created with errors, you can easily delete them using the registry editor. If the program under test is restarted while the registry editor is loaded, then, to update the information in the registry editor window, just press the F5 key.

21.6. Restoring information from the Windows registry

Modify the Form1_Load method:

```
private void Form1_Load(object sender, EventArgs e)
{
  RegistryKey rk = null;
  string path = "";
  int fileIndex = 0;
  try
  {
    rk = Registry.CurrentUser.OpenSubKey(regKeyName);
    if (rk != null)
    {
      Width = (int)rk.GetValue("FormWidth", Width);
      Height = (int)rk.GetValue("FormHeight", Height);
      splitContainer1.SplitterDistance =
        (int)rk.GetValue("Split1",
        splitContainer1.SplitterDistance);
      splitContainer2.SplitterDistance =
        (int)rk.GetValue("Split2",
        splitContainer2.SplitterDistance);
      comboBox1.SelectedIndex = (int)rk.GetValue("Zoom",
        comboBox1.SelectedIndex);
      path = (string)rk.GetValue("Path", "");
      fileIndex = (int)rk.GetValue("File", 0);
    }
  }
  finally
  {
    if (rk != null)
      rk.Close();
  }
  if (!Directory.Exists(path))
    path = Directory.GetCurrentDirectory();
  var node = InitialExpanding(Directory.GetCurrentDirectory());
```

```
  var node = InitialExpanding(path);
  if (node != null)
    treeView1.SelectedNode = node;
  if (fileIndex >= 0 && fileIndex < listBox1.Items.Count)
    listBox1.SelectedIndex = fileIndex;
}
```

Result. When the program starts, the data from the Windows registry is used to restore previously saved state of the program. If there is no data in the registry, then the default settings are used. Since the state of the file and directory structure may have changed since the last time the data was saved in the registry, we check the existance of the directory with the name obtained from the the **Path** registry value.

Disadvantage. If the form has been resized, it is not centered on screen on the next launch. This is because the actions to center the form are performed *before* the Load event occurs, that is, when the form has its original size.

Correction. In the Form1_Load method, before the statement

```
if (!Directory.Exists(path))
  path = Directory.GetCurrentDirectory();
```

add the following statements:

```
Left = (Screen.PrimaryScreen.WorkingArea.Width - Width) / 2;
Top = (Screen.PrimaryScreen.WorkingArea.Height - Height) / 2;
```

Result. Centering now works correctly for any form size.

Comments

1. The GetValue method of the RegistryKey class is used to read the data from the registry subkey values; in this case, it is enough to open the required subkey for reading only. The most convenient version of the GetValue method is with two parameters: the first parameter contains the registry value name, the second contains the default value (the method returns a default value if the specified registry value is not found in the registry subkey). The version of the method with one parameter (the regisrty value name) is less convenient, because, in the absence of the required regisrty value, it returns null. The GetValue method returns result of object type, so its return value must be explicitly cast to the type of the required data (int or string).

2. Registry editor is especially useful when debugging a program fragments responsible for reading data from the registry. With its help, you can delete some values of a registry subkey and the subkey itself, as well as make changes to the values, which allows you to test the program working in special situations.

3. To explicitly calculate the position of the form ensuring its centering, we used the standard Screen class, which allows us to obtain various characteristics of all screens associated with a computer. The PrimaryScreen property is used to

access the properties of the main screen, and the WorkingArea property returns a part of the screen without the Windows taskbar).

There is another way to correct the noted disadvantage. You may read the data from the registry *in the form constructor*. In this case, the information from the registry about the new form size will be available when centering the form.

22. MDI application: JPEGVIEW project

The JPEGVIEW project is a Multi-Document Interface (MDI) application. We describe methods of interaction between the main and child forms and standard actions on child forms (various types of placement of child forms on the main form, displaying a list of child forms in the menu, closing all child forms at the same time, etc.). We also discuss how to implement image scaling modes and keyboard scrolling for child forms.

22.1. Opening and closing child forms in MDI application

After creating the JPEGVIEW project, place non-visual controls of MenuStrip and OpenFileDialog type on Form1 (these controls will be named menuStrip1 and openFileDialog1; they will be placed in the non-visual control area under the form image; in addition, the menu associated with the menuStrip1 control will be indicated at the top of the form).

Using the menu designer (see Section 12.1), create a first-level menu item with the text **&File** in the menuStrip1 menu and use the **Properties** window to change the name of this item (that is, the Name property) to file1. In the drop-down menu associated with the **File** item, create a menu item with the text **&Open**, then an item with the text - (dash; this item will be converted to a horizontal separator), then an item with the text **E&xit**. Set the properties of Form1, its controls, and menu items added to the menuStrip1 menu (see also Fig. 22.1).

Properties (Form1)

```
Form1: Text = JPEG View, IsMdiContainer = True,
  StartPosition = CenterScreen
openFileDialog1: Title = Open image file,
  Filter = JPEG Images|*.jpg; *.jpeg
Open (the File group): Name = open1
Exit (the File group): Name = exit1
```

Add a new form to the project (it will be named Form2) and place the menuStrip1 menu control and the pictureBox1 container control for images on Form2. In the menuStrip1 menu of Form2, create a first-level menu item with the text **&File** and the name file1. In the drop-down menu associated with the **File** item, create two menu items with the text **&Open** and **&Close**, then an item with the text - (dash), then an item with the text **E&xit**. Set the properties of Form2, its controls, and menu items added to the menuStrip1 menu (see also Fig. 22.2).

Properties (Form2)

```
Form2: Text = empty string, ShowInTaskbar = False,
```

```
  AutoScroll = True
pictureBox1: SizeMode = AutoSize, Location = 0; 0,
  Modifiers = Internal
File: MergeAction = Replace
Open (the File group): Name = open1
Close (the File group): Name = close1
Exit (the File group): Name = exit1
```

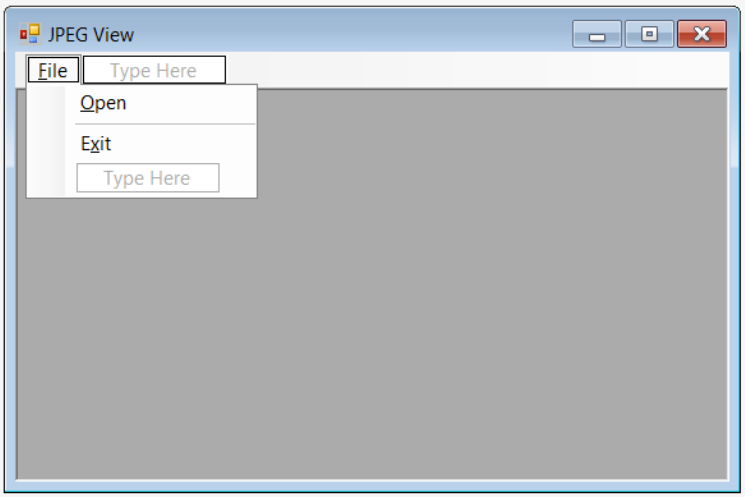


Fig. 22.1. Form1 view at the initial stage of development

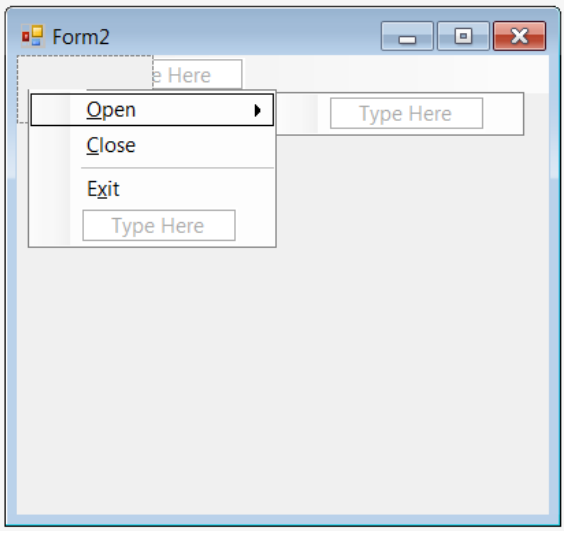


Fig. 22.2. Form2 view at the initial stage of development

Define event handlers for the Click event for the open1 and exit1 menu items of Form1:

open1.Click and exit1.Click handlers (Form1 menu items)

```
internal void open1_Click(object sender, EventArgs e)
// access modifier changed to internal
{
  openFileDialog1.FileName = "";
```

```
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    // create a child form and customize it
    Form2 f = new Form2();
    string n = openFileDialog1.FileName;
    Image i = Image.FromFile(n);
    f.pictureBox1.Image = i;
    f.Text = string.Format("{0} ({1}x{2})",
      System.IO.Path.GetFileNameWithoutExtension(n),
      i.Width, i.Height);
    f.MdiParent = this;
    f.Show();
  }
}
private void exit1_Click(object sender, EventArgs e)
{
  Close();
}
```

Define event handlers for the Click event for the open1, close1, and exit1 menu items of Form2:

open1.Click, close1.Click and exit1.Click handlers (Form2 menu items)

```
private void open1_Click(object sender, EventArgs e)
{
  (MdiParent as Form1).open1_Click(this, null);
}
private void close1_Click(object sender, EventArgs e)
{
  // close the child form
  Close();
}
private void exit1_Click(object sender, EventArgs e)
{
  // close the main form
  MdiParent.Close();
}
```

Result. Using the **Open** menu item in the *main form* **JPEG View**, you can load image files in JPEG format. Each of the uploaded files is displayed in a separate window (a *child window* or a *child form*), with the file name and image size in pixels (for example, **200x350**) indicated in the title bar of the child form. The initial size and position of the child form is determined by default (since its StartPosition property is equal to WindowsDefaultLocation). If the child

form is not large enough to display the entire image, then scroll bars appear in it. When the child form is maximized, its title is added to the title of the main form. When a child form is minimized, its title bar is placed at the bottom of the main form. If at least one child form is open, the additional **Close** command appears in the main form menu, which allows you to close the *active* child form. If all child forms are closed, the initial main form menu is restored.

To close the active child-form, you can use the Ctrl+F4 key combination. In addition, there are key combinations for activating the next and previous child forms: these are, respectively, Ctrl+F6 and Ctrl+Shift+F6. You can also switch between open child forms using the arrow keys. All these shortcut keys are processed automatically in the MDI application. See also Comments 1–4.

Disadvantage. If the width of the child form exceeds the size of the picture, then a menu bar is visible on the top of the child form.

Correction. For the menuStrip1 control of Form2, set the Visible property to **False**.

Result. Now the menu bar on child forms is not displayed. This does not affect the appearance of menu items added from this menu to the menu of the main form.

Comments

1. To access the main form, the MdiParent property of the child form is provided. Any form with a non-empty MdiParent property value is considered a child form. To define a form as the main form of an MDI application, it is enough to set its MdiContainer property to **True**.

2. The GetFileNameWithoutExtension method of the Path class from the System.IO namespace extracts the file name without the path and extension from the full file name. Since there is no using directives for this namespace in the **Form1.cs** file, we had to use the fully qualified class name: System.IO.Path.

3. Since the InitialDirectory property is not explicitly set for the openFileDialog1 control, the first time the program is started, its working directory will be selected as the initial directory (in our case, the **bin\Debug** subdirectory of the directory containing the JPEGVIEW project). If we select a file from another directory using the openFileDialog1 dialog box, then the next time the dialog box opens, this directory will be shown as the initial one. Information about the last used directory is automatically saved in a special section of the Windows registry, and this directory will be shown as the initial one when the program is started again. Such behavior of the program is quite reasonable and does not require any effort from the programmer to implement it. You can compare this approach with an explicit setting of the InitialDirectory property in the TEXTEDIT1 project (see Section 12.2).

4. A child form has no menu. When a child form is activated, the contents of its menuStrip1 control *replaces* the first-level menu items of the main form with the same names. In our case, the **File** menu item is replaced. The method

of replacement is determined by the MergeAction property of the child form menu item: in our case, it is equal to **Replace**, so the **File** menu item of the main form is replaced with the corresponding item of the child form. It should be noted that the AllowMerge property must be equal to **True** (this is the default value) for menuStrip1 controls of both main and child forms to be able to merge menus.

22.2. Standard actions with child forms

Create a new first-level menu item with the text **&Window** in the menuStrip1 menu of Form1 and use the **Properties** window to change the name of this item (that is, the Name property) to window1. In the drop-down menu associated with the **Window** item, create menu items with the text **&HTile**, **&VTile**, **&Cascade**, **&Arrange Icons** and set the Name properties of these items to hTile1, vTile1, cascade1, arrangeIcons1, respectively (see Fig. 22.3).

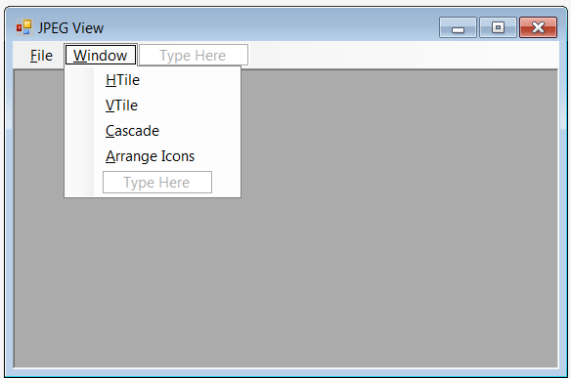


Fig. 22.3. Form1 view at an second stage of development

Add new statements to the constructor of the Form1 class:

```
public Form1()
{
  InitializeComponent();
  hTile1.Tag = MdiLayout.TileHorizontal;
  vTile1.Tag = MdiLayout.TileVertical;
  cascade1.Tag = MdiLayout.Cascade;
  arrangeIcons1.Tag=MdiLayout.ArrangeIcons;
}
```

Define the Click event handler for the hTile1 menu item, then connect the created handler to the Click event of the vTile2, cascade1, and arrangeIcons1 menu items.

hTile1.Click handler

```
private void hTile1_Click(object sender, EventArgs e)
{
  LayoutMdi((MdiLayout)(sender as ToolStripMenuItem).Tag);
}
```

Result. Using the commands of the **Window** group menu, you can perform standard actions on child forms.

- **HTile** and **VTile**. All child forms are arranged with no overlapping and so that they cover the entire client area of the main form. These commands differ in the way child forms are positioned: **HTile** places forms from top to bottom (stretching them horizontally), **VTile** places forms from left to right (stretching them vertically). Sometimes these commands work the same way (for example, for four or five child forms).
- **Cascade**. All child forms are arranged so that the title bars of all forms are visible on the screen. A unified size is set for all child forms; this size depends on the current size of the main form.
- **Arrange Icons**. All minimized child forms are located near the bottom border of the main form.

See also Comments 1–2.

Disadvantage. In the absence of child forms, the commands of the **Window** menu group lose their meaning, but are still available in the menu.

Correction. For the window1 menu item of Form1, set the value of the Modifiers property to **Internal** and the value of the Visible property to **False**.

In the open1_Click method of Form1, before the statement

```
f.Show();
```

insert a new statement:

```
window1.Visible = true;
```

Define the FormClosed handler for Form2:

Form2.FormClosed handler

```
private void Form2_FormClosed(object sender,
  FormClosedEventArgs e)
{
  if (MdiParent.MdiChildren.Length == 1)
  // there is only one child-form left,
  // which is currently being closed
    (MdiParent as Form1).window1.Visible = false;
}
```

Result. The **Window** menu item is displayed only if there is at least one open child form. See also Comments 3–4.

Comments

1. All actions on child forms are implemented in one method named LayoutMdi of the Form class; the action required is determined by its single parameter of the MdiLayout enumerated type. To make it possible to use single hTile1_Click handler for all menu items in the **Window** group, we assign the corresponding element of the MdiLayout enumeration to the Tag property of each menu item.

2. The hTile1_Click handler twice performs a type conversion: the first time for the sender parameter and the second time for the Tag property. In the first case, the as operator is used; in the second case, the (*type*) operator is used. The as operator can only be used to convert *reference* data types (for example, controls); this operator cannot be used for enumerated types and numeric data types. The (*type*) operator can be used to convert any data types, but it is less convenient to use for controls than the as operator. This is due to the fact that the operators of type conversion have a lower priority than the dot operator "." of access to a member of an object. For example, if we want to convert the sender parameter to the ToolStripMenuItem type and then use its Tag property, then the expression (ToolStripMenuItem)sender.Tag will be erroneous because it means that the sender.Tag property should be converted to the ToolStripMenuItem type. The ((ToolStripMenuItem) sender).Tag expression is correct, but looks worse than (sender as ToolStripMenuItem).Tag. The speed of execution of these expressions is practically the same. Recall that, when these operators are applied to *reference* data, the only difference between the (*type*) operator and the as operator is that, if the specified conversion is impossible, the (*type*) operator throws an exception of InvalidCastException type, whereas the as operator simply returns null.

3. The Form2_FormClosed method uses the MdiChildren property of Form[ ] type. This property contains all currently open child forms. In this case, we were only interested in the *number* of these forms, which can be obtained using the Length property of the MdiChildren array.

4. Notice that the first time we access the MdiParent object in the Form2_FormClosed method, we do not need to convert it to the Form1 type, because the Form class already has the MdiChildren property. When accessing the window1 menu item, which is a field of the Form1 class (and *is absent* in the Form class), the conversion of MdiParent to the Form1 type is required. Note also that, to be able to access the window1 field of Form1 from *another class* (namely, Form2), we need to change its Modifiers property to **Internal**.

22.3. Adding a list of open child forms to the menu

For the menuStrip1 control of Form1, set the value of the MdiWindowListItem property to **window1**.

Result. A list of all open child forms is now displayed in the **Window** menu group; a check mark is displayed near the active child form. By clicking on an item from this list, you can activate the corresponding child form. To select an item from the list, you can also press the key corresponding to its ordinal number (from 1 to 9). If more than 9 child forms are open, the **More Windows...** menu item is displayed below the item corresponding to the 9th child form. By selecting it, you can display an auxiliary **Select Window** dialog box with a list of all open child forms.

22.4. Closing all child forms at the same time

Add a new item to the **File** menu of Form2 with the text **Close &All**. This item must be added *after* the **Close** item; to do this, select the separator below the **Close** item, press the right mouse button, and select the **Insert | MenuItem** command from the context menu that appears (you can also do it differently: create a **Close All** menu item at the end of the **File** group and then drag it with the mouse to a new location). Set the Name property of the newly created menu item to closeAll1 and define the Click event handler for this menu item:

closeAll1.Click handler

```
private void closeAll1_Click(object sender, EventArgs e)
{
  foreach (Form f in MdiParent.MdiChildren)
    f.Close();
}
```

Result. When the **Close All** command is executed, all child forms are closed.

Remark. The **Close All** command is included in the Form2 menu because it should only be available if there are open child forms in the application.

22.5. Image scaling

Add a new item with the text **&Zoom** to the **File** menu of Form2. This item must be added after the **Close All** item using one of the methods described at the beginning of Section 22.4. Set the Name property of the newly created menu item to zoom1, set the CheckOnClick property to **True**, and define the CheckedChanged event handler for the menu item:

zoom1.CheckedChanged handler

```
private void zoom1_CheckedChanged(object sender, EventArgs e)
{
  if (zoom1.Checked)
  {
    pictureBox1.Dock = DockStyle.Fill;
    pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
  }
  else
  {
    pictureBox1.Dock = DockStyle.None;
    pictureBox1.SizeMode = PictureBoxSizeMode.AutoSize;
  }
}
```

Result. Images in the child forms can be scaled to fit the current size of the child form (while maintaining the aspect ratio of the image). To enable/disable

the scaling mode, the **Zoom** checkbox-like menu item is provided. If the zoom mode is enabled, a check mark is displayed near this menu item. When the zoom mode is disabled, the original image size is set; in this case, scroll bars may appear on the child form. Each child form sets the zoom mode independently; the state of the **Zoom** menu item corresponds to the zoom mode for the active child form. See also Comment 2 in Section 21.4.

22.6. Automatic resizing of child forms

Add two new items to the **File** menu of Form2 with the text **&Resize** and **R&esize All**. New items must be added after the **Zoom** item using one of the methods described at the beginning of Section 22.4. Set the Name properties of the created menu items to resize1 and resizeAll1, respectively. The final view of the **File** group menu for Form2 is shown in Fig. 22.4.

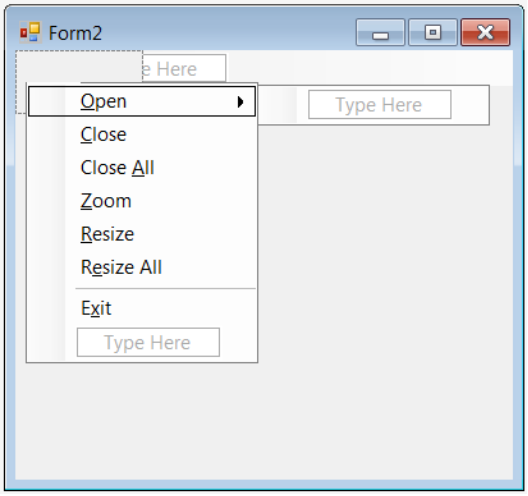


Fig. 22.4. The final view of Form2

Define event handlers for the Click event for the new menu items:

resize1.Click and resizeAll1.Click handlers

```
private void resize1_Click(object sender, EventArgs e)
{
  Size sz = ClientSize;
  if (zoom1.Checked)
  {
    int iw = pictureBox1.Image.Width,
      ih = pictureBox1.Image.Height,
      cw = ClientSize.Width,
      ch = ClientSize.Height;
    double x = cw / (double)iw,
      y = ch / (double)ih;
    if (x < y)
      sz.Height = (int)(x * ih);
```

```
    else
      sz.Width = (int)(y * iw);
  }
  else
    sz = pictureBox1.Size;
  ClientSize = sz;
}
private void resizeAll1_Click(object sender, EventArgs e)
{
  MdiParent.LayoutMdi(MdiLayout.Cascade);
  foreach (Form f in MdiParent.MdiChildren)
    (f as Form2).resize1_Click(f, null);
}
```

Result. The **Resize** command sets the size of the active child form equal to the current size of the image it contains. The **Resize All** command resizes and cascades all child forms.

Remark. Calling the LayoutMdi method that cascades child forms should be done before sizing, because the LayoutMdi method itself resizes the child forms.

Comment

If the zoom mode is disabled, then setting the size of the form is trivial: you just need to set the size of its client area equal to the size of the pictureBox1 control. You cannot do this in the zoom mode, because the size of the pictureBox1 control is *not* equal to the size of the scaled image. Moreover, it is impossible to determine what part of a control occupies an image in zoom mode. At the same time, in the zoom mode, it is sufficient to correct the size of the form in only *one* dimension (in width or height). To determine which dimension and by what amount to adjust, the proportions of the original (unscaled) image and the client area of the form are compared.

22.7. Additional control tools

Place a *toolbar* on Form1, that is, a control of ToolStrip type (this control will be named toolStrip1). The toolbar will automatically dock to the top border of the form and will be positioned below the menuStrip1menu.

Proceeding in the same way as in Section 15.1, add four ToolStripButton controls to the toolStrip1 toolbar, change their names (that is, Name properties) to the following: open2, close2, resize2, zoom2, and set the Text properties to **Open**, **Close**, **Resize**, and **Zoom**, respectively. For all four buttons, set the DisplayStyle property to **Text**, AutoSize to **False**, and Size.Width to **50** (the last two settings are needed to set the buttons to the same width, since by default the width of the buttons is determined by the size of their caption). The resulting toolbar is shown in Fig. 22.5.

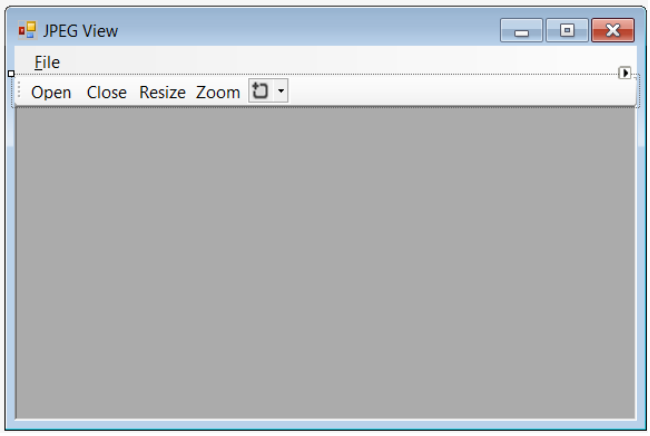


Fig. 22.5. The final view of Form1

Change the access modifiers for some members of Form2: for the zoom1 menu item, set the Modifiers property to **Internal**; for the zoom1_CheckedChanged and resize1_Click methods, replace the private modifier with internal in their headers.

Return to Form1 and connect the Click event of the open2 button to the existing open1_Click handler, and define the Click event handlers for the close2, resize2, and zoom2 shortcut buttons as follows:

close2.Click, resize2.Click, zoom2.Click handlers

```
private void close2_Click(object sender, EventArgs e)
{
  ActiveMdiChild.Close();
}
private void resize2_Click(object sender, EventArgs e)
{
  (ActiveMdiChild as Form2).resize1_Click(this, null);
}
private void zoom2_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = (ActiveMdiChild as Form2).zoom1;
  mi.Checked = !mi.Checked;
}
```

Result. The **Open** shortcut button duplicates the menu command of the same name. The **Close** and **Resize** buttons close and resize the active child form; the **Zoom** shortcut button toggles image scaling on and off in the active child form.

Remark. When implementing actions on clicking the **Close**, **Resize**, and **Zoom** buttons, the ActiveMdiChild property of the main form is used, which returns the active child form.

Error. Clicking the **Close**, **Resize**, or **Zoom** buttons with no child forms throws a NullReferenceException (because the ActiveMDIChild property is null in this case).

Correction. For the close2, resize2, and zoom2 buttons, set the Modifiers property to **Internal**, the Enabled property to **False**.

Add new statement to the open1_Click method of Form1 before the statement f.Show():

```
close2.Enabled = resize2.Enabled = zoom2.Enabled = true;
```

Change the Form2_FormClosed method of Form2:

```
private void Form2_FormClosed(object sender,
  FormClosedEventArgs e)
{
  Form1 f = MdiParent as Form1;
  if (MdiParent.MdiChildren.Length == 1)
    (MdiParent as Form1).window1.Visible = false;
    f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
      f.zoom2.Enabled = false;
}
```

Result. If the main form contains no child forms, then the **Close**, **Resize**, and **Zoom** buttons are unavailable.

Disadvantage 1. When you hover the mouse cursor over any shortcut button, a tooltip appears next to it duplicating the text on the button.

Correction. In our case, there is no need to display tooltips, since the buttons themselves contain text that explains their purpose. Therefore, we may disable the display of tooltips for the *entire toolbar* by setting the ShowItemToolTips property of the toolStrip1 control to **False** (see also Comment 1).

Disadvantage 2. Unlike the **Zoom** menu item, the view of the **Zoom** shortcut button does not allow to determine whether the active child form is in zoom mode.

Correction. Define an Enter event handler for Form2:

Form2.Enter handler

```
private void Form2_Enter(object sender, EventArgs e)
{
  (MdiParent as Form1).zoom2.Checked = zoom1.Checked;
}
```

Add the following statement to the zoom1_CheckedChanged method:

```
(MdiParent as Form1).zoom2.Checked = zoom1.Checked;
```

Change the last statement in the Form2_FormClosed method as follows:

```
f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
  f.zoom2.Enabled = f.zoom2.Checked = false;
```

Remark. The change in the Form2_FormClosed method ensures that an inaccessible **Zoom** button will never be in the pressed state.

Result. Now the **Zoom** shortcut button is in pressed state if the zoom mode is set for the active child form. The **Zoom** button changes state in the following situations (see also Comment 2):

- when clicking this button;
- when executing the **Zoom** menu command;
- when switching to a child form with the other zoom mode.

Comments

1. If you want to disable the tooltip for a particular ToolStrip button, set the AutoToolTip property to **False** for this item and check that its ToolTipText property is empty. Note that, if AutoToolTip is **True** (which is the default value for the ToolStripButton control), then the ToolTipText property always has the same value as the Text property.

2. Note that the zoom2 button itself does not change its state when clicked; it only changes the state of the zoom1 menu item, but this change calls the zoom1_CheckedChanged handler, which changes the state of the button. This is the simplest way of interaction between a button and a menu item, in which the state of the button can change both when you click on it and when the associated menu command is executed.

22.8. Scrolling the image using the keyboard

Define an event handler for the KeyDown event for Form2:

Form2.KeyDown handler

```
private void Form2_KeyDown(object sender, KeyEventArgs e)
{
  if (e.Modifiers != Keys.Control)
    return;
  Point p = AutoScrollPosition;
  Size s = pictureBox1.Size;
  p.X = -p.X;
  p.Y = -p.Y;
  switch (e.KeyCode)
  {
    case Keys.Up:
      p -= new Size(0, 10); break;
    case Keys.Down:
      p += new Size(0, 10); break;
    case Keys.Left:
      p -= new Size(10, 0); break;
    case Keys.Right:
```

```
      p += new Size(10, 0); break;
    case Keys.Home:
      p = Point.Empty; break;
    case Keys.PageDown:
      p = new Point(s); break;
    case Keys.End:
      p = new Point(0, s.Height); break;
    case Keys.PageUp:
      p = new Point(s.Width, 0); break;
  }
  AutoScrollPosition = p;
}
```

Result. Image scrolling (if the zoom mode is disabled and the child form contains only part of the image) can be performed not only with the scroll bars, but also using the keyboard while holding down the Ctrl key: the arrow keys provide scrolling in the specified direction, and the keys Home, End, PgUp, and PgDn provide movement to one of the corners of the image (to the upper left, lower left, upper right and lower right, respectively). The need to use the Ctrl key is explained by the fact that ordinary arrow keys are already reserved in the MDI application for switching between child forms (see Section 22.1).

Let us give an example of a running program (Fig. 22.6). The program window contains two child forms with the same image. In the left-hand (active) form, the zoom mode is enabled, the right-hand form has scroll bars.

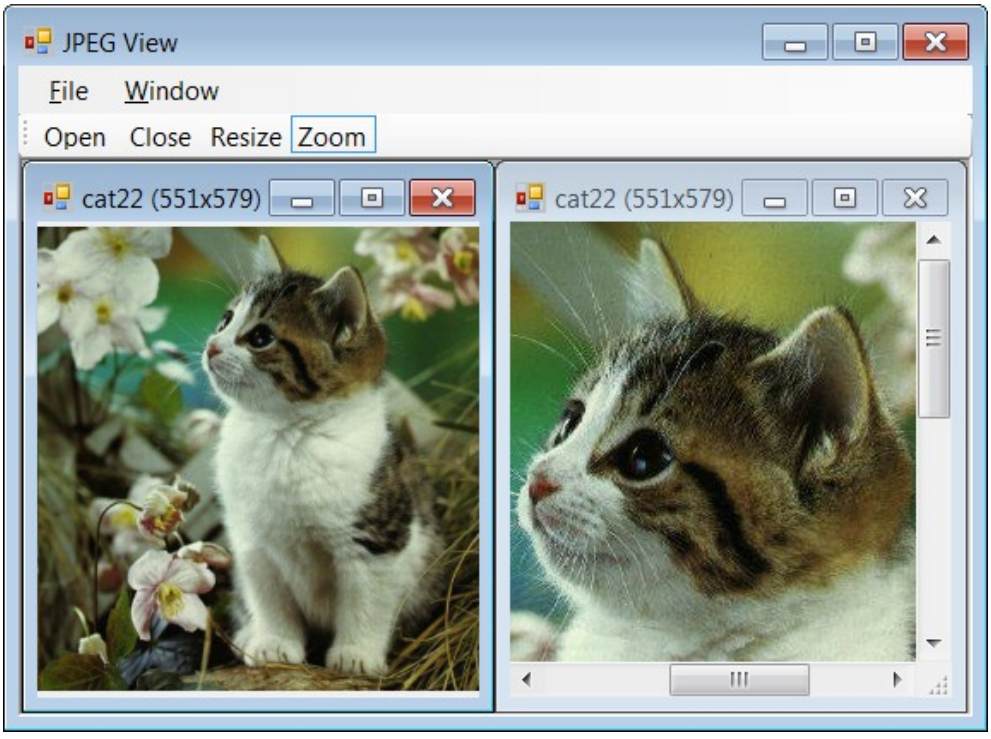


Fig. 22.6. View of the running JPEGVIEW application

Comment

The AutoScrollPosition property of a form (and of any visual control that is a descendant of the ScrollableControl class, for example, Panel) allows you to programmatically control the scrolling of its contents.

This property, like the SelectedText property of the TextBox control (see Comment 2 in Section 8.1), belongs to a group of properties that do not behave quite naturally: if you set the AutoScrollPosition property to the value of some object p of Point type with positive fields X and Y (which are the positions of the scroll bars) and then read the new value of this property, you get an object of Point type with *negative* fields X and Y (more precisely, the field values will be *opposite* of p.X and p.Y). Therefore, for our program to work properly, we must use the p.X = –p.X and p.Y = –p.Y statements at the beginning of the Form2_KeyDown method.

23. Splash screen application: TRIGFUNC project

The TRIGFUNC project applies the DataGridView control to show multicolumn list of data. It also describes techniques related to processing command line arguments, displaying a splash window with special properties, and early termination of the program. It uses the Sleep method of the Thread class, the DoEvents method of the Application class, and the ProgressBar control.

23.1. Creating a table of trigonometric function values

After creating the TRIGFUNC project, place the DataGridView table control on Form1 (this control will be named dataGridView1) and set the properties of the form and the added control:

Properties

```
Form1: Text = Trigonometric Functions,
  StartPosition = CenterScreen
dataGridView1: Dock = Fill, AllowUserToResizeColumns = False,
  AllowUserToResizeRows = False, AutoSizeColumnsMode = Fill,
  RowHeadersVisible = False, ReadOnly = True
```

In addition to setting the properties of the dataGridView1 table, you must set the properties of its columns. To do this, select the Columns property in the **Properties** window of the dataGridView1 control and click the ellipsis button ... near this property. The **Edit Columns** dialog box appears allowing you to create table columns, define their order, and set their properties. Click the **Add…** button in this window; in the **Add Column** window that appears, type the header of the first column in the **Header** text box: **x** (no other settings need to be changed), and click the **Add** button. An image of the created column will appear on the left-hand side of the **Edit Columns** window, and a table that allows you to set column's properties will appear on the right-hand side (the structure of this table is similar to the structure of the **Property** window). Add four more columns to the dataGridView1 table using similar actions and specifying the following headers for them: **Sin(x * pi)**, **Cos(x * pi)**, **Tan(x * pi)**, **Cot(x * pi)**. Then close the **Edit Columns** window by clicking the **OK** button. As a result, the dataGridView1 control will display the headers of the created columns (see Fig. 23.1). See also Comments 1–2.

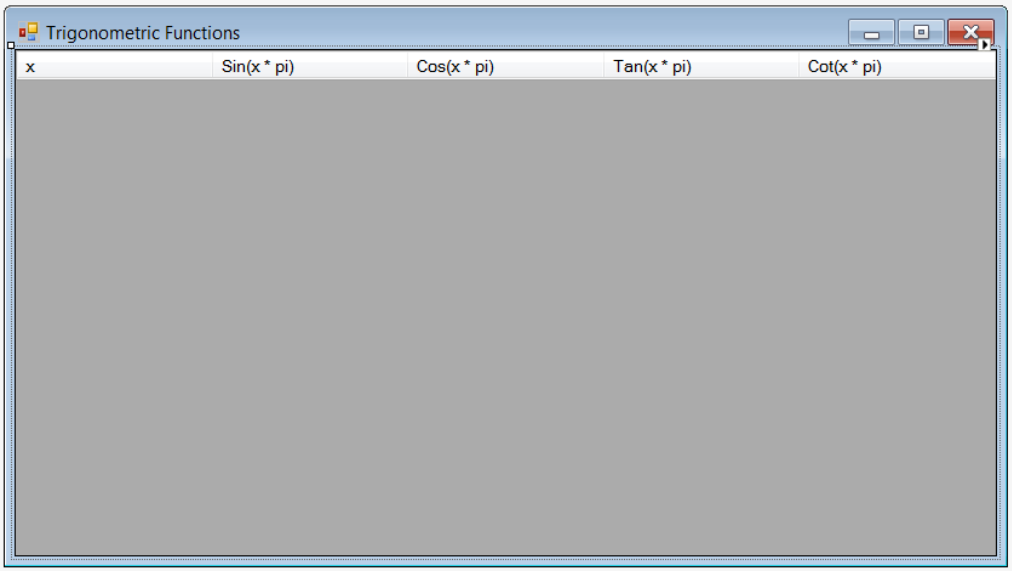


Fig. 23.1. Form1 view at the initial stage of development

Define a handler for the Load event of Form1:

Form1.Load handler

```
private void Form1_Load(object sender, EventArgs e)
{
  int n = 7, nMax = 100001;
  string[] args = Environment.GetCommandLineArgs();
  if (args.Length > 1)
    try
    {
      int n0 = int.Parse(args[1]);
      if (n0 < 2 || n0 > nMax)
        throw new Exception();
      else
        n = n0;
    }
    catch
    {
      string s = string.Format("Invalid parameter: {0}\n" +
        "Valid values are from 2 to {1}", args[1], nMax);
      MessageBox.Show(s, "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
      return;
    }
  double step = 1.0 / (n - 1);
  // reduce the width of the first column:
```

```
  dataGridView1.Columns[0].Width /= 2;
  // determine the number of rows:
  dataGridView1.RowCount = n;
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
  }
}
```

Result. The program fills the dataGridView1 table with the values of trigonometric functions with arguments from 0 to π radians on a grid of *n* equally spaced points. The point number *n* can be specified as a command line argument; valid values are from 2 to 100001. When starting the program from Visual Studio, you can specify the parameters in the **Command line arguments** text box of the **Debug settings** section in the project properties window (recall that this window is called by the **Project | <*Project name*> Properties** menu command). If the parameter is specified incorrectly, an error message is displayed and the program exits immediately. If the parameter is not specified, then the number of points is assumed to be 7. See also Comments 3–8.

Comments

1. Note that, when adding a new column, you can specify its *type*; this type allows you to determine which type will be used by default for all cells of a given column. In addition to the text displayed on the column header (**HeaderText**), you can specify the column name (**Name**), which is an identifier similar to the names of the controls placed on the form (by default, the names Column1, Column2, etc. are used). Finally, when you create a new column, you can immediately define three of its boolean properties: Visible (default value **True**), ReadOnly, and Frozen (default value **False**). Only the last property needs explanation: if Frozen is **True**, then this column and all the columns preceding it are "frozen", that is, they stop moving on the screen when scrolling horizontally in the DataGridView table. All of these properties, as well as a number of others, can be defined after creating a column using the table of properties in the **Edit Columns** window.

2. It is possible to define columns of the DataGridView table and set their properties without visual tools. The easiest way to do this is to use the Add

method of the Columns property of the DataGridView control. This method has two versions: the first version has two string parameters columnName and headerText, this version allows you to create a column with text data (that is, a column of DataGridViewTextBoxColumn type); the second version has a parameter of DataGridViewColumn type, this parameter can be a column object of any possible type. The Add method returns the index of the created column. In our case, taking into account that all columns are of DataGridViewTextBoxColumn type and their names will not be used in the future, we could simply put the following five statements in the constructor of the Form1 class (instead of the steps described above for setting columns):

```
dataGridView1.Columns.Add("", "x");
dataGridView1.Columns.Add("", "Sin(x * pi)");
dataGridView1.Columns.Add("", "Cos(x * pi)");
dataGridView1.Columns.Add("", "Tan(x * pi)");
dataGridView1.Columns.Add("", "Cot(x * pi)");
```

3. To access the command line arguments, the GetCommandLineArgs method of the Environment class was used, which returns a string array. The first element of this array (with index 0) contains the full name of the application executable file. Starting with the next element, the array contains command line arguments.

4. In our case, the command line argument may be invalid for two reasons: firstly, it may be a string that cannot be converted to an integer; secondly, even if such a conversion is possible, the resulting number may be out of range. In order to perform error handling in one place in the program, we implement this error handling in the catch clause of the try block containing statement that converts the command line argument to an integer. If such a conversion fails, an exception is raised and the catch clause is immediately activated. If the conversion is successful, but the number is out of range, then, to activate the catch section, the program explicitly throws an exception using the throw operator (the type of the exception does not matter in this case, so an exception of Exception type is thrown). For details on handling exceptions, see Chapter 3.

5. For immediate termination of the program, we call the Close method of the form. Note that, even after calling the Close method, the current method will run to the end. Therefore, after calling Close, you must immediately exit the current method using the return statement.

6. Due to the **Fill** value of the AutoSizeColumnsMode property of the dataGridView1 control, the columns "stretch" to the width of the control. By default, the width of all columns is the same, but you can change it for any column. For example, in our program, the width of the first column has been halved in the Form1_Load method.

7. When performing calculations with double type, you do not have to worry about possible overflow. In particular, when calculating the value of the

function Cot at zero, no error will occur and the value **infinity** (or **бесконечность**) will be displayed in the corresponding cell of the table. See also Comment 2 in Section 3.1.

8. When converting the obtained function values to their string representation by means of the ToString method, the default numeric format (the *general format*) is used, in which the shortest possible representation of the given number is selected. Any other formatting option must be explicitly specified as a parameter to the ToString method. For example, for the **x** column, the program uses the f5 format that is a format with a fixed number of fractional digits (in this case, with *five* fractional digits).

23.2. Displaying the splash window when loading the program

Add a new form to the project (it will automatically be named Form2) and place label1 on Form2. Set the properties of Form2 and label1. To set the specified properties (Name, Size and Bold) of the Font property of label1, it is convenient to display the **Font** dialog box by clicking the ellipsis button ... near the Font property in the **Properties** window. Using the **Font** window, you can configure all the font properties and, in addition, you can view a sample font with the selected settings.

Properties

```
Form2: Text = empty string, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%,
  ShowInTaskbar = False, StartPosition = CenterScreen,
  UseWaitCursor = True
label1: Text = Trigonometric Functions,
  AutoSize = False, Dock = Fill, TextAlign = MiddleCenter,
  Font.Name = Times New Roman, Font.Size = 32, Font.Bold = True
```

After setting the Font property of the label1 control, resize the Form2 so that the label text is placed on two lines (see Fig. 23.2).

Fig. 23.2. Form2 view at the initial stage of development

Remark. Recall that, to select a form in a situation when one of its controls is selected, you can just press the Esc key. Another quick way to select a form (clicking on its title bar) is inapplicable in this case, because, due to the settings made earlier, Form2 has no title bar.

Add a new field to the beginning of the Form1 class description in the **Form1.cs** file:

```
private Form2 form2 = new Form2();
```

Change the constructor for the Form1 class as follows:

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  form2.Show();
  Application.DoEvents();
}
```

Add a new statement to the Form1_Load method:

```
form2.Hide();
```

Result. While loading the Form1 main window and filling the dataGridView1 table with data, a Form2 *splash window* is displayed on the screen indicating that the program is currently performing initializing actions. The splash window has no title and is semi-transparent; it cannot be moved around the screen. On the splash window, the mouse cursor changes to an hourglass. When the main window is displayed, the splash window disappears.

Disadvantage. If the amount of calculations required to fill the table is small (for example, when using the default number of points $n = 7$), then the splash window cannot be viewed due to the short display time.

Correction. In the Form1_Load method, before the previously added statement form2.Hide(); insert a new statement:

```
System.Threading.Thread.Sleep(1000);
```

Result. After the completion of the initializing actions, the program pauses for 1 second, during which the splash window remains on the screen.

Comments

1. The Opacity property of the Form class is responsible for the transparency mode of the form. By default, it is **100%**, which corresponds to the *full opacity* mode. Partial transparency mode is convenient for dialog windows if, when they are displayed on the screen, it is desirable to see the text of the main window located under them (note that such dialog windows are used in the Visual Studio environment when displaying exception messages). In our case, we used a semi-transparent splash window just to give it a more original look.

2. In the absence of the Application.DoEvents() statement in the Form1 constructor, an *empty rectangular area* would be displayed instead of a splash window. Calling the DoEvents method of the Application class provides immedi-

ate handling of all events that have occurred up to that point, in particular, events related to the redrawing of controls on the screen. More precisely, the DoEvents method provides processing of *all Windows messages* currently contained in the message queue for this application (recall that, in .NET applications, Windows messages are converted to events for certain controls).

3. To suspend program execution for the specified number of milliseconds, we use the Sleep method of the Thread class from the System.Threading namespace.

23.3. Using the splash window as an information window

Place button1 on Form2 and set the properties of Form2 and the added control (see also Fig. 23.3).

Properties

```
Form2: AcceptButton = button1
button1: Text = OK, DialogResult = OK, Visible = False
```

Fig. 23.3. Form2 view at intermediate stage of development

Place menuStrip1 on Form1. If the menu bar added to Form1 is overlapped by the upper part of the dataGridView1 table, then select the menuStrip1 control and click on the **Send to Back** button on the right side of the **Layout** panel (see the comment in Section 21.3).

Using the menu designer (see Section 12.1), create a first-level menu item with the text **&About…** in the menuStrip1 menu (see Fig. 23.4) and use the **Properties** window to change the name of this item (that is, the Name property) to about1. Define the Click event handler for the created menu item:

about1.Click handler

```
private void about1_Click(object sender, EventArgs e)
{
  form2.ShowDialog();
}
```

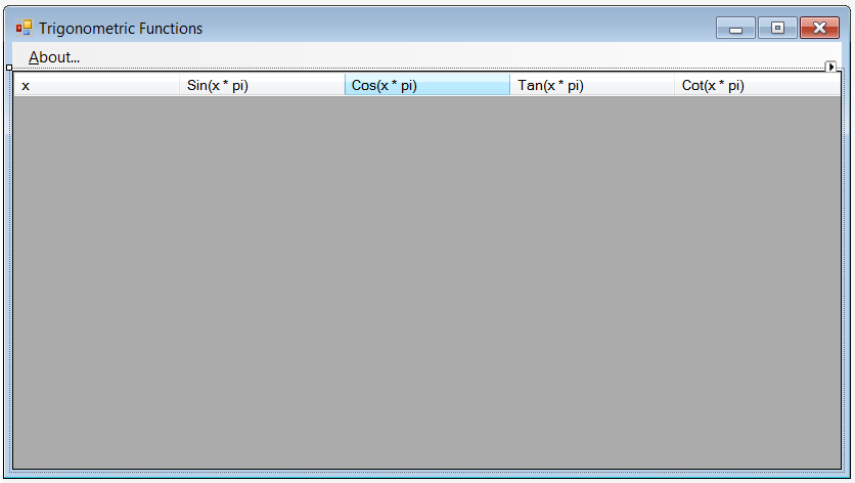


Fig. 23.4. The final view of Form1

Add two statements to the Form1_Load method:

```
form2.UseWaitCursor = false;
form2.Controls["button1"].Visible = true;
```

Result. Now the splash window can be displayed (as a modal window) after loading the main form using the **About…** menu command. To close the splash window in this case, use the **OK** button or the Enter key. When the splash window is displayed during loading of the main form, the **OK** button is not displayed on it.

23.4. Displaying the progress of the program loading

Place the ProgressBar control (named progressBar1) on Form2 and set its properties (see also Fig. 23.5):

Properties

```
progressBar1: Dock = Bottom, Modifiers = Internal,
  Style = Continuous
```

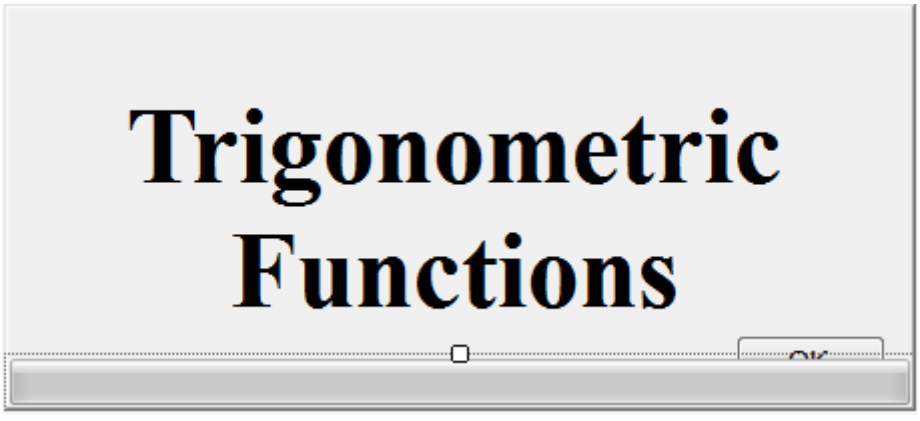


Fig. 23.5. The final view of Form2

Modify the Form1_Load method of Form1 as follows:

```
private void Form1_Load(object sender, EventArgs e)
{
```

```
  int n = 7, nMax = 100001;
  string[] args = Environment.GetCommandLineArgs();
  if (args.Length > 1)
    try
    {
      int n0 = int.Parse(args[1]);
      if (n0 < 2 || n0 > nMax)
        throw new Exception();
      else
        n = n0;
    }
    catch
    {
      string s = string.Format("Invalid parameter: {0}\n" +
        "Valid values are from 2 to {1}", args[1], nMax);
      MessageBox.Show(s, "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
      return;
    }
  double step = 1.0 / (n - 1);
  dataGridView1.Columns[0].Width /= 2;
  dataGridView1.RowCount = n;
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
    form2.progressBar1.Value = 100 * i / (n - 1);
  }
  System.Threading.Thread.Sleep(1000);
  form2.Hide();
  form2.UseWaitCursor = false;
  form2.Controls["button1"].Visible = true;
  form2.progressBar1.Visible = false;
}
```

Result. When the main window is loading, the splash window displays graphical information about the loading progress (that is, about the percentage of calculations performed); the progressBar1 control is used for this. On subsequent calls of the splash window using the **About…** menu item, the progressBar1 control is not displayed.

Comments

1. The ProgressBar control implements the *progress bar* and allows you to display information about the progress of a certain process in three modes determined by the Style property: the indication in the form of a *continuous strip* (**Continuous**, used in our program), indication in the form of a *set of rectangular blocks* (**Blocks**, the default value), and indication in the form of *running markers* (**Marquee**). The latter mode is convenient to use in a situation where you need to indicate the execution of some long-term actions, but it is difficult to determine which part of the actions has already been completed. In **Marquee** mode, no additional customization of the ProgressBar is required; just note that the MarqueeAnimationSpeed property can be used to specify the *speed* at which markers move. In the other two modes, the appearance of the indicator depends on the values of three integer properties: Value, Minimum (default value **0**), and Maximum (default value **100**). The value of the Value property ranges from Minimum to Maximum and determines which part of the indicator will be highlighted. Although the ProgressBar control provides the Increment and PerformStep methods to change the Value property, there is no particular need for these methods because the Value property is mutable.

2. By setting the value of the Modifiers property of the progressBar1 control to **Internal**, we got the opportunity to directly access this control from the methods of another form (namely, the Form1 owner form). In this case, there is no need to use the Controls collection property (which was used in Section 23.3 to access from Form1 to the button1 control of Form2).

23.5. Early termination of the program

Define a handler for the KeyDown event for Form2:

Form2.KeyDown handler

```
private void Form2_KeyDown(object sender, KeyEventArgs e)
{
  if (e.KeyCode == Keys.Escape)
    Hide();
}
```

In the Form1_Load method of Form1, at the beginning of the for loop, insert a new piece of code:

```
Application.DoEvents();
if (!form2.Visible)
{
```

```
  Close();
  return;
}
```

Result. The program execution can be interrupted during the initial calculations; to do this, just press the Esc key. In this case, the splash window disappears from the screen and the program exits immediately.

Comments

1. Pay attention to calling the DoEvents method before checking if form2 remains visible on the screen. As noted earlier (see Section 23.2), the DoEvents method allows you to handle events that occurred after the start of some method (in our case, after the start of the Form1_Load method, we want to handle event related with pressing the Esc key). In the absence of the DoEvents method, all such events will be handled only *after* the current method finishes, that is, in our case, after the completion of *all* actions associated with the calculation of data.

2. When the splash window is displayed using the **About...** command, pressing the Esc key does not close it. This is due to the fact that, in this case, the events associated with the keyboard go directly to the active control button1, bypassing the Form2. In order for the Esc key to close the window in this case, it is enough to set the KeyPreview property of Form2 to **True**. After the correction has been made, the splash window called up by the **About...** command can be closed either by pressing the Enter key, or by pressing the Esc key. Instead of setting the KeyPreview property to **True**, you can set the CancelButton property to **button1**.

23.6. Dragging the splash window

Add the field to the Form2 class declaration:

```
private Point p;
```

Define event handlers for the MouseDown and MouseMove events for the label1 control of Form2:

label1.MouseDown and label1.MouseMove handlers

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
}
private void label1_MouseMove(object sender, MouseEventArgs e)
{
  if (Control.MouseButtons == MouseButtons.Left)
    Location += new Size(e.X - p.X, e.Y - p.Y);
}
```

Result. The splash window can now be dragging around the screen with the mouse; to do this, place the mouse cursor in any position of the window, press

the left mouse button, and, without releasing it, move the mouse to the required position on the screen.

Note. Previously, a similar drag method was used in Section 9.1. This method is most natural for those windows that do not contain a title bar.

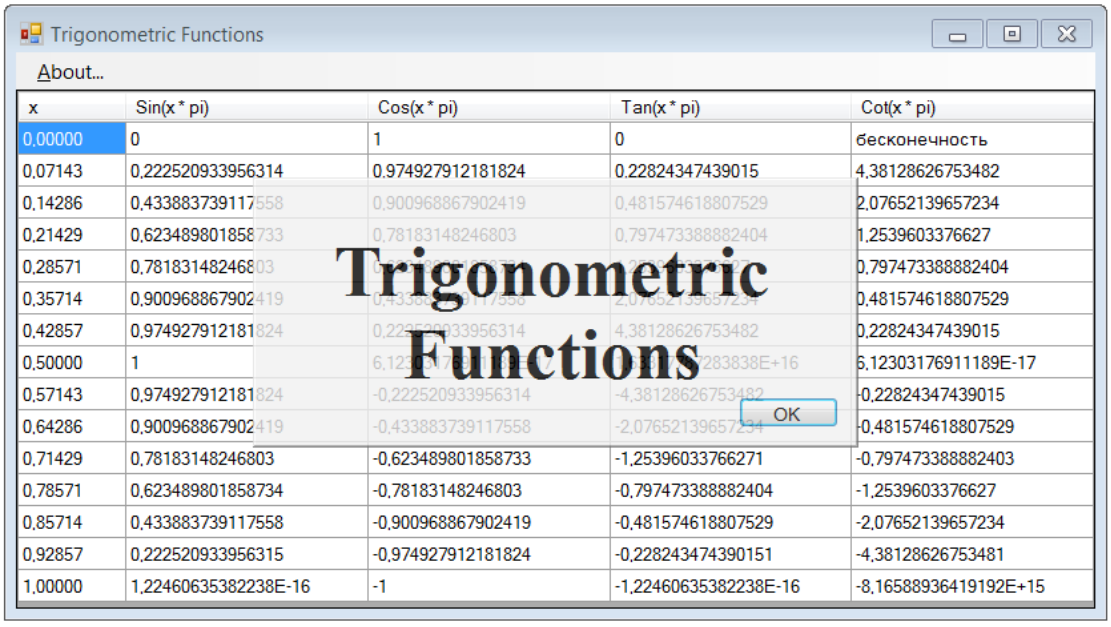
Disadvantage. The splash window can also be dragged when it is first appears on the screen (during data initialization), which is usually not desirable.

Correction. Select the label1 control and *clear* the text box associated with the MouseMove event in the **Properties** window. In the declaration of the label1_MouseMove method, replace the private modifier with internal. In the **Form1.cs** file, add a new statement to the Form1_Load method:

```
form2.Controls["label1"].MouseMove += form2.label1_MouseMove;
```

Result. Dragging the splash window is possible only when it is displayed on the screen using the **About…** command. At the moment of initial loading of the program, the splash window cannot be dragged, because the label1_MouseMove method is not yet associated with the MouseMove event of the label1 control.

Let us give a view of the running program after calling the splash window by the **About…** command in the case when the number of points *n* is 15 (Fig. 23.6).



x	Sin(x * pi)	Cos(x * pi)	Tan(x * pi)	Cot(x * pi)
0,00000	0	1	0	бесконечность
0,07143	0,222520933956314	0,974927912181824	0,22824347439015	4,38128626753482
0,14286	0,433883739117558	0,900968867902419	0,481574618807529	2,07652139657234
0,21429	0,623489801858733	0,78183148246803	0,797473388882404	1,2539603376627
0,28571	0,781831482468			0,797473388882404
0,35714	0,900968867902419			0,481574618807529
0,42857	0,974927912181824	0,222520933956314	4,38128626753482	0,22824347439015
0,50000	1	6,12	283838E+16	6,12303176911189E-17
0,57143	0,974927912181824	-0,222520933956314	-4,38128626753482	-0,22824347439015
0,64286	0,900968867902419	-0,433883739117558	-2,07652139657234	-0,481574618807529
0,71429	0,78183148246803	-0,623489801858733	-1,25396033766271	-0,797473388882403
0,78571	0,623489801858734	-0,78183148246803	-0,797473388882404	-1,2539603376627
0,85714	0,433883739117558	-0,900968867902419	-0,481574618807529	-2,07652139657234
0,92857	0,222520933956315	-0,974927912181824	-0,228243474390151	-4,38128626753481
1,00000	1,22460635382238E-16	-1	-1,22460635382238E-16	-8,16588936419192E+15

Fig. 23.6. View of the running TRIGFUNC application

24. Creating controls at runtime: HTOWERS project

The HTOWERS project is a computer implementation of the well-known logic problem "Towers of Hanoi" (or "Tower of Hanoi"). This problem is as follows: there are three rods, one of them contains (in decreasing order of size) several disks forming a "tower"; it is required to move the entire tower to one of the empty rods using the other empty rod as an auxiliary one. You can move one disk at a time; a larger disk cannot be placed on a smaller one. For solving the problem with N disks, the minimum number of moves is $2^N - 1$ (see [4, Chapter 1]).

The HTOWERS application uses three rectangular areas (GroupBox controls) instead of rods and rectangular blocks (Label controls) instead of discs; blocks can be moved between areas. The HTOWERS project demonstrates the capabilities associated with creating controls at runtime. Moving blocks is implemented using the drag-and-drop mode. The application has a demo mode showing the correct sequence of moves to solve the problem.

24.1. Creating a start position

After creating the HTOWERS project, place three controls of GroupBox type (they will be named groupBox1 – groupBox3) and the NumericUpDown control (named numericUpDown1) on Form1. Set the properties of Form1 and all the added controls and arrange the controls as shown in Fig. 24.1.

Properties

```
Form1: Text = Towers of Hanoi, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
groupBox1: Text = Start position
groupBox2-groupBox2: Text = empty string
numericUpDown1: Minimum = 2, Maximum = 10, Value = 4
```

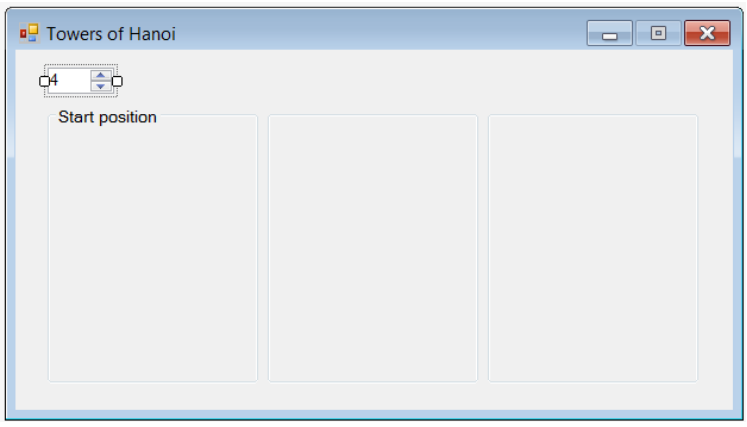


Fig. 24.1. Form1 view at the initial stage of development

Define the Load event handler for Form1:

Form1.Load handler

```
private void Form1_Load(object sender, EventArgs e)
{
  Random r = new Random();
  // n is the number of tower blocks:
  int n = (int)numericUpDown1.Value,
    w = groupBox1.Width,
    h = groupBox1.Height;
  for (int i = 0; i < n; i++)
  {
    Label lb = new Label();
    // defining the properties of the created Label control
    // (the i-th block of the tower);
    // all blocks are drawn on the groupBox1 control:
    lb.Parent = groupBox1;
    lb.BorderStyle = BorderStyle.FixedSingle;
    lb.Size = new Size((w - 10) * (n - i) / n, (h - 15) / n);
    lb.Location =
      new Point((w - lb.Width) / 2, h - 2 - (i + 1) * lb.Height);
    // the label background color is determined randomly:
    lb.BackColor = Color.FromArgb(r.Next(256), r.Next(256),
      r.Next(256));
  }
}
```

Result. When the program starts, a tower of four colored blocks (which are the controls of Label type) is drawn on the groupBox1 control.

Comment

Any container control (in particular, a form) can be specified as the *parent* control of any visual control by assignment to Parent property of this visual control. Instead of the lb.Parent = groupBox1 statement, we can use the equivalent call to the Add method for the Controls collection property of the parent control:

```
groupBox1.Controls.Add(lb);
```

The parent control is responsible for redrawing all of its child controls. If the control has no parent, it cannot be displayed on the screen.

24.2. Redrawing the tower when changing the number of blocks

Define a handler for the ValueChanged event for the numericUpDown1 control:

numericUpDown1.ValueChanged handler

```
private void numericUpDown1_ValueChanged(object sender,
  EventArgs e)
{
  Form1_Load(this, null);
}
```

Result. Changing the value of the numericUpDown1 control creates a new tower with the specified number of blocks.

Error. The new tower is drawn over the old one.

Correction. Add a new method label_Dispose to the declaration of the Form1 class:

```
private void label_Dispose(GroupBox gb)
{
  for (int i = gb.Controls.Count - 1; i >= 0; i--)
    gb.Controls[i].Dispose();
}
```

Add the following statement at the beginning of the numericUpDown1_Value-Changed method:

```
label_Dispose(groupBox1);
```

Result. Now all blocks of the old tower disappear from the screen and also free their resources.

Disadvantage. In the process of forming a new tower, extraneous fragments of images flicker on the screen. This diadvantage is due to the fact that after defining a parent for a label, this label is immediately drawn on its parent, and all subsequent changes to its properties lead to redrawing of the label.

Correction. In the Form1_Load method, move the statement

```
lb.Parent = groupBox1;
```

to the end of the loop body (that is, to the position after the statement defining the background color of the label).

Result. Now all settings of the label's properties are performed *before* its parent control is defined, therefore, they do not lead to redrawing of the label on the screen (the label is drawn only once after setting all its properties).

24.3. Dragging blocks to a new location

Add a new statement to the constructor of the Form1 class:

```
groupBox1.AllowDrop = groupBox2.AllowDrop =
  groupBox3.AllowDrop = true;
```

Add new methods named label_MouseDown and label_Move to the Form1 class:

```
private void label_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
```

```
    DoDragDrop(sender, DragDropEffects.Move);
}
private void label_Move(Label lb, GroupBox gb)
{
  // change the GroupBox control that displays the lb label:
  lb.Parent = gb;
  lb.Top = gb.Height - 2 - gb.Controls.Count * lb.Height;
}
```

Add the following statement at the end of the loop body in the Form1_Load method:

```
lb.MouseDown += label_MouseDown;
```

Define event handlers for the DragEnter and DragDrop events for the groupBox1 control and then connect these handlers to the DragEnter and DragDrop events of the groupBox2 and groupBox3 controls:

groupBox1.DragEnter and groupBox1.DragDrop handlers

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void groupBox1_DragDrop(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  GroupBox gb = sender as GroupBox;
  if (gb == lb.Parent)
    return;
  label_Move(lb, gb);
}
```

Result. Any block (that is, the Label control) can be dragged to another tower (that is, the GroupBox control), and the moved block will always be at the top of the tower.

Remark. Actions related to moving a label to another GroupBox control are implemented as a special label_Move method, since these actions will be performed not only when executing DragDrop handlers, but also in an additional demo mode of the program (see Section 24.6).

Error. You can drag not only the top block, but also any other block of the tower.

Correction. Change the groupBox1_DragEnter method as follows:

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb)
```

```
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Result. Only the top block of the tower can be dragged. When you try to drag the lower blocks, the drag-and-drop cursor takes the form of a prohibition sign ⃠.

Remark. We took advantage of the fact that, when you add a child to the Controls collection, it is placed at the end of it; therefore, the top block has an index equal to Controls.Count – 1 in the Controls collection.

It remains to take into account the additional condition of the problem: *the block can move either to an empty tower or to a tower with a top block of the larger size*. To do this, we change the groupBox1_DragEnter method again:

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  int k = int.MaxValue;
  GroupBox gb = sender as GroupBox;
  if (gb.Controls.Count > 0)
    k = gb.Controls[gb.Controls.Count - 1].Width;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb
    || lb.Width > k)
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Result. When dragging the block, the additional condition is taken into account.

Remark. The new version of the groupBox1_DragEnter method uses the variable k, which contains the width of the top block of the receiving panel or the maximum possible int value (equal to int.MaxValue) if the receiving panel contains no blocks.

Error. When changing the number of blocks using the numericUpDown1 control, only those blocks that are on the groupBox1 panel are deleted.

Correction. Change the numericUpDown1_ValueChanged method:

```
private void numericUpDown1_ValueChanged(object sender,
  EventArgs e)
{
  label_Dispose(groupBox1);
  label_Dispose(groupBox2);
  label_Dispose(groupBox3);
```

```
  Form1_Load(this, null);
}
```

Comment

Drag-and-drop mechanism was discussed in detail in Chapter 10. Note that the AllowDrop property for the GroupBox controls had to be defined in the form's constructor, since this property is not displayed in the **Properties** window. Next, note that all Label controls have their AllowDrop property set to **False** (the default value), so labels cannot act as drop receivers (they are "invisible" to the drag-and-drop mode). Therefore, the drag-and-drop cursor does not have a prohibition sign over the labels, and if you drop the source object over the label, the DragDrop event will occur for the GroupBox control containing this label.

24.4. Restoring the start position and counting the number of block movings

Place the Button control (named button1) on Form1, set the Text property of the button to **Start position**, and connect its Click event to the existing numericUpDown1_ValueChanged handler. Also, place the Label control (named label1) on Form1 (you do not need to change its Text property). Arrange the added controls according to Fig. 24.2.

Fig. 24.2. Form1 view at an intermediate stage of development

Add declarations of two fields and the Info helper method to Form1 class:

```
private int count;
private int minCount;

private void Info()
{
  label1.Text = $"Number of moves: {count} ({minCount})";
}
```

Add three statements to the Form1_Load method:

```
count = 0;
minCount = (int)Math.Round(Math.Pow(2, n)) - 1;
Info();
```

Add two statements to the label_Move method:

```
count++;
Info();
```

Result. To restore the starting position with the same number of blocks, press the **Start position** button. If, when restoring the initial position, it is necessary to change the number of blocks, it is sufficient to specify a new value in the numericUpDown1 control (in this case, you do not need to press the button).

Information about the number of block movings is displayed in the text of label1. There, in parentheses, the minimum number of moves required to solve the problem with a given number n of blocks is indicated (this minimum number is equal to $2^n - 1$).

Remark. Note that block movings within the same GroupBox control are not counted when calculating the number of moves.

Comments

1. When forming the text of label1 in the Info method, an *interpolated string* was used (see Comment 7 in Section 2.2).

2. The Pow function of the Math class was used to find the value of 2^n. Since it returns a result of double type, the resulting value must be converted to an integer type. Since the fractional part is discarded when using the (int) operator, we first round the resulting number to the nearest integer using the Round function. Note that the (int) conversion remains necessary after rounding, since the Round function returns a result of double type (with a zero fractional part).

24.5. Information about solving the problem

Place another label (named label2) on Form1 under the existing label, set its Text property to **Problem solved!** and set its ForeColor property to **Green**.

In the Form1_Load method, add the statement

```
label2.Visible = false;
```

Add the following piece of code to the label_Move method:

```
if (groupBox2.Controls.Count == numericUpDown1.Value
    || groupBox3.Controls.Count == numericUpDown1.Value)
  label2.Visible = true;
```

Result. The problem is considered solved and the message **Problem solved!** is displayed if the size of the tower at one of two final positions (on the groupBox2 or groupBox3 control) is equal to the total number of blocks.

Disadvantage. After solving the problem, dragging blocks is still allowed.

Correction. Add the following piece of code to the *beginning* of the groupBox1_DragEnter method:

```
if (label2.Visible)
```

```
{
  e.Effect = DragDropEffects.None;
  return;
}
```

Result. After solving the problem, blocks cannot be dragged.

24.6. Demo mode implementation

Place another button on Form1 (button2) and set its Text property to **Demo**. Form1 will take the view shown in Fig. 24.3.

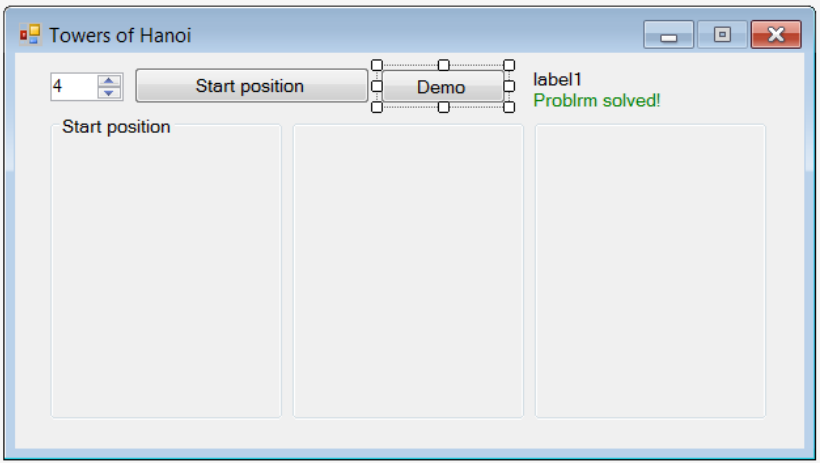


Fig. 24.3. The final view of Form1

Add a new Step method to the Form1 class:

```
private void Step(int n, GroupBox src, GroupBox dst,
  GroupBox tmp)
{
  if (n == 0)
    return;
  Step(n - 1, src, tmp, dst);
  if (button1.Enabled)
    return;
  label_Move(src.Controls[src.Controls.Count - 1] as Label, dst);
  Application.DoEvents();
  System.Threading.Thread.Sleep(1500 /
    ((int)numericUpDown1.Value) - 1);
  Step(n - 1, tmp, dst, src);
}
```

Define the Click event handler for button2:

button2.Click handler

```
private void button2_Click(object sender, EventArgs e)
{
  numericUpDown1.Enabled = button1.Enabled = !button1.Enabled;
```

```
  if (!button1.Enabled)
  {
    if (groupBox1.Controls.Count != numericUpDown1.Value)
      numericUpDown1_ValueChanged(null, null);
    Step((int)numericUpDown1.Value, groupBox1, groupBox3,
      groupBox2);
    numericUpDown1.Enabled = button1.Enabled = true;
  }
}
```

Add the following statement to the *beginning* of the label_MouseDown method:

```
if (!button1.Enabled)
  return;
```

Result. When you click the **Demo** button, the program switches to *demo mode*, which demonstrates the correct sequence of moves to solve the problem with a given number of blocks (blocks move automatically). In demo mode, the numericUpDown1 control and the **Start position** button are not available; also, you cannot drag labels. Demo mode is exited after completing the solution, as well as when the **Demo** button is clicked again (in the latter case, after exiting the demo mode, you can continue to solve the problem yourself).

Error. When trying to terminate a program in demo mode, an ArgumentOutOfRangeException run-time error occurs due to the fact that previously called Step methods continue execution after the Controls collection properties of the GroupBox controls have been cleared as a result of the terminating program actions.

Correction. Define the FormClosed event handler for Form1:

Form1.FormClosed handler

```
private void Form1_FormClosed(object sender,
  FormClosedEventArgs e)
{
  button1.Enabled = true;
}
```

Result. Now, when the form is closed, button1 is made available, which makes it possible to almost immediately complete all recursive calls to the Step method without accessing the Controls collection.

Comment

The Step(n, src, dst, tmp) method defines the actions required to solve a problem with n blocks. The last three parameters (of GroupBox type) define the starting area (the src parameter, *source*), the ending area (the dst parameter, *destination*), and the auxiliary area (the tmp parameter, *temporary*). Obviously, to solve a problem with n blocks, it is enough to perform three steps.

Step 1. Move n - 1 blocks from the src area to the tmp area using the dst area as an auxiliary.

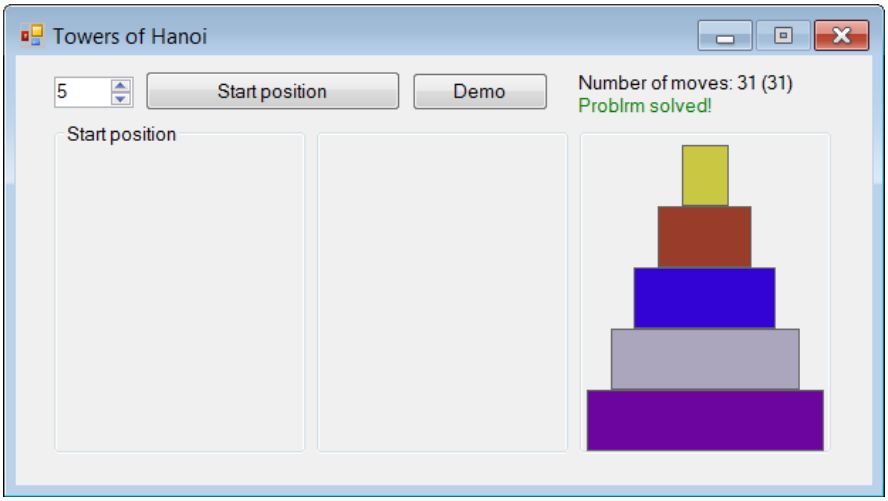
Step 2. Move the n-th block from the src area to the dst area.

Step 3. Move n - 1 blocks from the tmp area to the dst area using the src area as an auxiliary.

Steps 1 and 3 can me performed as a call to the same step method with the first parameter equal to n - 1. Thus, the method implements a *recursive algorithm*. The chain of recursive operations should be interrupted when the parameter n becomes equal to 0. In addition, the Step method provides another way to exit: if the **Demo** button was pressed again (to check this action, the Enabled property of the button1 control is analyzed).

Step 2 is implemented using the label_Move method. After calling this method, you must call the DoEvents method of the Application class, which provides redrawing the moved label and also allows to handle the **Demo** button click if the user wants to exit demo mode earlier. The Sleep method of the Thread class is also called, which pauses the program execution for a while (this period of time depends on the number of blocks). We used the DoEvents and Sleep methods earlier in Chapter 23 (see Comments 2–3 in Section 23.2 and Comment 1 in Section 23.5).

Here is a view of a running program after solving a problem with five blocks (Fig. 24.4).



Fig, 24.4. View of the running HTOWERS application

25. Study assignments

25.1. General requirements

If the form does not contain controls for which it makes sense to resize, then it cannot be resized or maximized.

When placing controls in a form, make optimal use of the form space. For forms of variable size, you should actively use the Anchor or Dock properties.

All actions provided in the project must be associated with keyboard shortcuts. Information about keyboard shortcuts should be available in the program window through underlined characters in the titles of controls, additional text in parentheses (for example, **Run (F5)**), default buttons. Changing the focus with the Tab key should occur in a natural order: left to right and top to bottom.

Programs should not contain handlers of the same type; instead, you must connect one handler to multiple controls. Actions applied to multiple controls must be performed in a loop using the Controls property of the form.

The default names of controls should not be changed, except for the names of menu items and shortcut buttons.

25.2. CONSOLE project: console applications, file and directory processing

General guidelines. All input data for the program must be passed using command line arguments. To test a program with different arguments, use the project settings specified in the **Debug** group and prepare a special directory structure on disk.

The program should provide formatted output (for instance, using interpolated strings). LINQ queries can be useful in a number of situations.

The program must correctly handle directories with relative paths (that is, with paths that do not start with a drive letter and a backslash). For example, specifying the **dir1** directory means that the **dir1** directory is a subdirectory of the current directory. Special directory names "." (one dot) and ".." (two dots) do not need to be processed.

At the beginning of its work, the program should print to the console an example of its call with the command line arguments (if some arguments are optional, they are enclosed in square brackets). If there is an invalid argument (for example, the name of a directory that does not exist), the program should display an error message. After the completion of the data output, it is necessary that the form does not close immediately and allows the user to view the results obtained.

1. At the end of program work, a list of files from the current directory (or the directory specified as a command line argument) is displayed in the console. Files are sorted by size. Information about files should be displayed in three columns containing the file name, size (in bytes) and creation date. You do not need to process subdirectories.

2. At the end of program work, a list of subdirectories of the current directory (or the directory specified as a command line argument) is displayed in the console. Subdirectories of all levels must be listed in the order of their nesting, and directories must be indented with four spaces for each level. At each level, the directories must be sorted alphabetically by name.

3. At the end of program work, the console displays a list of files from the current or user-specified directory with names matching the user-specified mask (for example, ***.jpg**). The directory and mask are specified by the user on the command line (the first is the mask, the second is the directory). If the second argument is missing, the current directory is processed. If all arguments are missing, an error message is displayed. Information about files should be displayed in three columns containing the file name, its size (in bytes) and the date of creation (files are sorted by name in alphabetical order). You do not need to process subdirectories.

4. At the end of program work, the console displays a list of files matching the specified mask in all-level subdirectories of the current directory (or the directory specified as the first command line argument). The mask is specified as the second argument; if it is not specified, then it is considered equal to ***.***. For each subdirectory, you should output its full name and then a list of the names of the files that satisfy the mask and are located in this subdirectory, together with their size in bytes (each name is displayed on a new line with an indention of 4 spaces, the files are sorted in alphabetical order of names). Subdirectories can be listed in any order.

5. At the end of program work, the console displays summary information about files matching the specified mask in all-level subdirectories of the current directory (or the directory specified as the first command line argument). The mask is specified as the second argument; if it is not specified, then it is considered equal to ***.***. For each subdirectory, first its name is displayed, then (in the same line) the number of files matching the mask and their total size in bytes. Subdirectories of all levels must be listed in the order of their nesting, directories must be indented with four spaces for each level. Within each level, directories can be listed in any order.

6. At the end of program work, the console displays a list of files from the user-specified directory with names that match the user-specified creation date interval (subdirectories are not processed; the start date and end date may coin-

cide). The directory, the start date, and the end date are specified by the user on the command line as three arguments. If there are less than three command line arguments, an error message is displayed. Information about files should be displayed in three columns containing the file name, its size (in bytes), and the creation date. Files must be sorted in ascending order of creation date. In an example of program call (this example should be print to the console at the beginning of program work), the required date format should be displayed.

7. Create a program that compares the contents of files with the same name (case insensitive) in the two directories specified by the user as two command line arguments. If there are less than two command line arguments, an error message is displayed. Two lists of files are displayed on the screen: (1) a list of files with the same name and the same contents, (2) a list of files with the same name and different contents. In each list, files must be sorted alphabetically, with each name displayed on a new line.

8. Create a program that compares the contents of two directories specified by the user as two command line arguments. If there are less than two command line arguments, an error message is displayed. Three lists of files must be displayed on the screen: (1) a list of files present in the first directory and absent in the second, (2) a list of files present in the second directory and absent in the first, (3) a list of files contained in both directories. File contents is not required to analyze. In each list, files must be sorted alphabetically, with each name displayed on a new line. Filename comparisons are not case sensitive. You do not need to process subdirectories.

9. Create a program that copies the structure of the nested directories. In the created directory structure, each subdirectory name must have a prefix specified by the user in the first command line argument. The source directory (containing the directory structure to be copied) and the destination directory (in which the copy of this structure will be created) must be specified as the second and third command line argument, respectively. If there are less than three command line arguments, an error message is displayed. The program must display the number of created directories. You do not need to copy files.

10. Create a program that calculates two values: the number and total size of all files located in the specified directory and its all-level subdirectories and satisfying the mask specified by the user. The top-level directory and mask are specified by the user as the first and second command line arguments, respectively. If no mask is specified, the ***.*** mask is used. If the directory is also not specified, then the current directory is processed. Several masks can be specified in the command line arguments; in this case, each mask is processed separately, and the program displays more than two values (each two values must be displayed on a new line).

11. Create a program that searches for a file by mask in the structure of nested subdirectories, excluding some subdirectories from consideration. The initial directory name, file mask, and subdirectory names to be excluded are specified by the user on the command line (there can be multiple names of excluded subdirectories). Files of the initial directory are always processed. For all found files, their full name, size (in bytes), and creation date are displayed (files can be displayed in any order). If names of excluded directories are not specified, then all subdirectories are processed; if, in addition, no file mask is specified, then all files are displayed. If all command line arguments are missing, the current directory is processed.

12. Create a program that finds the characteristics of the structure of nested directories: the total number of subdirectories, the maximum nesting depth, the maximum number of files in one directory, the total size of all files in the initial directory and all subdirectories. The initial directory is specified as a command line argument; if the initial directory is not specified, then the current directory is processed.

13. Create a program to rename all files with a user-specified name in a given directory and all its subdirectories. The user specifies the old and new file names (without * and ? wildcards) and the top-level directory as command line arguments. If there are less than three command line arguments, an error message is displayed. Searching for files to rename should not be case sensitive. The program displays the total number of renamed files.

25.3. DIALOGS project: form interaction

General guidelines. For dialog boxes (that is, modal forms), it is necessary to implement the standard actions for the Enter and Esc keys (the Enter key is always associated with the **OK** button or its analog, the Esc key is always associated with the **Cancel** key or its analog). When you reopen the dialog box, it must either retain the previous information or display new information if such is the condition of a study assignment. In any case, it is necessary to ensure that the first control of the dialog box is activated. The dialog box should not contain minimize and maximize buttons. The main form must contain an available minimize button; maximization should only be available for the resizable main form. By default (if a study assignment does not require the dialog box to be resizable), the dialog box should be fixed in size.

1. The fixed-size main form contains a text box, the **Password** label, and the **Open a protected form** button. The initial password is **qwerty**. If the password is input correctly, then, when the button (or the Enter key) is pressed, a modal form with the **Protected form** title appears containing two text boxes with the **New password** label (at the beginning, these text boxes contain the initial password) and two modal buttons: **OK** and **Cancel**. The **OK** button is avail-

able if both text boxes in the modal form contain the same non-empty text. When you close the modal form with the **OK** button or the Enter key, this text becomes the new password. When you close the modal form with the **Cancel** button or the Esc key, the password is not changed. When input a password, the symbols "*" should be displayed instead of the typed characters; to do this, use the PasswordChar property of the TextBox control. You do not need to save your password when you close the application.

2. The fixed-size main form contains the **Change scaling** button (the button is located at the upper left corner of the form). When the button is pressed, a modal form with the **Scaling** title appears containing a text box with the **New scale (%)** label and three buttons: **OK**, **Apply**, and **Cancel**. The **OK** and **Apply** buttons are available if the text box contains an integer in the range from **10** to **300**. The initial value of this input box is **100**; only digits can be input in this box. When you press the **OK** or **Apply** buttons, the main form and the button it contains change their sizes in accordance with the new scale factor (for example, if the factor is **200**, then the sizes are doubled); in case of pressing the **OK** button, the modal form will close. The scaling is always relative to the initial size of the form. When you click the **Cancel** button, the modal form closes without performing any additional action. When you reopen the modal form, the text box should display the current scale factor value.

3. The resizable main form contains a multi-line text box. When you try to close the form, a modal form appears with buttons: **Save**, **Not save**, **Cancel**. When you click the **Save** button, the entered text will be saved in the **text.txt** file. Pressing the **Cancel** button cancels the closing of the main form. When you reopen the form, if there is a **text.txt** file, its text is loaded into the text box.

4. The resizable main form contains the **Show** button. When this button is clicked, a modal form appears with two text boxes containing the current values of the coordinates of the upper left corner of the **Show** button (the **x** coordinate is displayed in the first text box, the **y** coordinate in the second) and the buttons **OK**, **Apply**, and **Cancel** . When you click the **OK** or **Apply** buttons, new coordinates are set for the **Show** button; in case of pressing the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action. If the text boxes contain invalid values, the **OK** and **Apply** buttons should be unavailable (the value is considered valid if it can be converted to a non-negative number and, after moving to the specified position, the **Show** button will be at least partially visible on the screen).

5. The fixed-size main form contains the **Twin** button. When you click the button, a modal form of the same size appears with the **OK** and **Cancel** buttons. The modal form can be resized. The **OK** button (and the Enter key) closes the

modal form and sets the size of the modal form for the main form, the **Cancel** button (and the Esc key) also closes the modal form, but does not change the size of the main form.

6. The resizable main form contains the **Add Label** button. When you click the button, a modal form appears with two text boxes for the coordinates of a new label, a text box for the text of this label, and buttons **OK**, **Apply**, and **Cancel**. The **OK** and **Apply** buttons are available if the coordinates of a new label are non-negative numbers corresponding to some point in the main form and the label text is a non-empty string. When you click the **OK** or **Apply** buttons, a new label is created in the main form in the position specified by the user; in case of clicking the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action. If the modal form was closed by pressing the **OK** button, then, when it is reopened, the coordinates for the new label should increase by **20**.

7. The fixed-size main form contains the **Toss a coin** button and labels displaying statistics with the text **Total number of tosses = 0** and **Percentage of heads = 0**. When you click the **Toss a coin** button, a modal form appears with a text box for input the number of tosses and the **OK**, **Apply**, and **Cancel** buttons. The **OK** and **Apply** buttons are available if the text box contains an integer in the range from **1** to **10000**. The initial value of this input box is **10**; only digits can be input in this box. When you press the **OK** or **Apply** buttons, the required number of coin tosses is simulated and statistics labels are updated in the main form; in case of pressing the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action. Use the Random class to simulate a coin toss.

8. The fixed-size main form contains three labels. When you click on one of the labels, a modal form appears with a text box and buttons **OK**, **Apply**, and **Cancel**. The text box contains the text of the label being clicked; this text can be changed. When you click **OK** or **Apply**, the text of the corresponding label in the main form is updated; in case of pressing the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action.

9. The fixed-size main form contains three text boxes and a label (not a button!) with the text **Change edit mode**. At the launching of the program, the first text box has focus, and all text boxes are editable. When you click on the label, a modal form appears with the label **Are you sure?** and buttons **Yes** and **No**. Clicking the **Yes** button changes the edit mode for the text box that has focus (editable mode is switched to read-only mode and vice versa). Pressing the **No** button does not change the edit mode. In any case, the modal form will close.

You should implement the modal form yourself without using the MessageBox class.

10. The fixed-size main form contains the **Change Background** button. When you click this button, a modal form appears with three text boxes for input the intensities of the red, green, and blue color components. In addition, the modal form contains buttons **OK**, **Apply**, and **Cancel**. When you click the **OK** or **Apply** buttons, the background color of the main form changes; in case of clicking the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action. If at least one of the text boxes contains text other than a number from the range 0–255, then the **OK** and **Apply** buttons should be unavailable. Use the Color structure and its FromRGB method.

11. The resizable main form contains the **Change alignment** button. When you click on this button, a modal form appears with two drop-down lists labeled **Horizontal alignment** and **Vertical alignment**. Each list contains three options: **Align Left**, **Align Center**, **Align Right** for the **Horizontal alignment** list and **Align Top**, **Align Middle**, **Align Bottom** for the **Vertical alignment** list. In addition, the modal form contains buttons **OK**, **Apply**, and **Cancel**. When you click the **OK** or **Apply** buttons, the main form changes its position on the screen in accordance with the values of the drop-down lists; in case of clicking the **OK** button, the modal form will close. When you click the **Cancel** button, the modal form closes without performing any additional action. When aligning, take into account the current size of the main form. Use the PrimaryScreen.WorkingArea property of the Screen class. Working with drop-down lists is described in the LISTBOXES project (Section 19.1).

25.4. SYNC project: control synchronization

General guidelines. In all projects, it is required to configure all controls of the same type by specifying *one* event handler for each group of controls of the same type. If statements (such as if (n == 1), if (n == 2), if (n == 3), ...) or switch statements should not be used in handlers. It is allowed to use the Tag properties of the controls, as well as the Controls property of the form, which allows you to refer to a control by its name. See the CALC project (Section 6.1) for information on sharing event handlers.

Working with *text boxes* is described in the CALC project (Section 6.5) and TEXTBOXES project (Section 8.1). Working with *checkboxes* is described in the CHECKBOXES project (Chapter 20). Working with *radio buttons* is described in the TEXTBOXES project (Section 8.2). Working with *toolbar* and *shortcut buttons* is described in the TEXTEDIT4 project (Chapter 15). Working with *track bars* is described in the COLORS project (Section 18.1). Working with NumericUpDown controls is described in the TEXTEDIT6 project (Sec-

tion 17.3). Working with *progress bars* is described in the TRIGFUNC project (Section 23.4).

1. The main form contains six text boxes with the text **1** – **6** and the **Show** button. When the **Show** button is clicked, a second (non-modal) form appears containing six checkboxes (the CheckBox controls). The checkboxes are labeled **1** – **6**; initially none of them is checked. When you check any checkbox, the text of the corresponding text box is highlighted in bold; when you uncheck the checkbox, the bold highlighting of the corresponding text box is canceled. When changing the text of any text box, the label of the corresponding checkbox should change accordingly.

2. The main form contains six text boxes with the text **1** – **6** and the **Show** button; the text of the first text box must be in bold. When the **Show** button is clicked, a second (non-modal) form appears, containing six radio buttons (the RadioButton controls). Radio buttons are labeled **1** – **6**; the radio button corresponding to the bold text box must be selected. When you select another radio button, the bold highlighting is transferred to the text of the corresponding text box. When changing the text of any text box, the label of the corresponding radio button should change accordingly.

3. The main form contains the **Show** button and the toolbar (the ToolStrip control) with 6 shortcut buttons (the ToolStripButton controls). The shortcut buttons on the toolbar have titles **1** – **6**; initially, none of the shortcut buttons is in the pressed state. When the **Show** button is clicked, a second (non-modal) form appears containing 6 checkboxes (the CheckBox controls). The checkboxes are labeled **1** – **6**. When checking/unchecking any checkbox, the corresponding shortcut button on the toolbar is automatically pressed/released; when the shortcut button is pressed/released on the toolbar, the corresponding checkbox is automatically checked/unchecked.

4. The main form contains a **Show** button and the toolbar (the ToolStrip control) with 6 shortcut buttons (the ToolStripButton controls). The shortcut buttons on the toolbar have titles **1** – **6**, one of the shortcut buttons is in the pressed state (initially it is the shortcut button **1**). When the **Show** button is clicked, a second (non-modal) form appears containing 6 radio buttons (the RadioButton controls). Radio buttons are labeled **1** – **6**; the radio button corresponding to the pressed shortcut button in the main form should be selected. When another radio button is selected, the corresponding shortcut button is automatically pressed (and the previously pressed button is released); when the shortcut button is pressed, the corresponding radio button is automatically selected (and the previously pressed shortcut button is released).

5. The main form contains the **Show** button and six red labels with the text **Color**. When the **Show** button is clicked, a second (non-modal) form appears

containing six panels (the Panel controls); each panel contains three radio buttons. Radio buttons have labels **Red**, **Green**, **Blue** (these labels can be made common for all panels by placing three additional Label controls to the left of the first panel). Initially, the **Red** radio button is selected in each panel. When you switch radio buttons, the text color of the corresponding label on the main form is corrected. When you click on any label of the main form, its color changes cyclically (from red to green, from green to blue, from blue to red) and the corresponding radio button is automatically selected on the corresponding panel of the second form.

6. The main form contains the **Show** button and seven NumericUpDown controls with the text **0**. When the **Show** button is clicked, a second (non-modal) form with seven track bars (the TrackBar controls) appears. When you move the slider of some track bar, the number in the corresponding text box should automatically change. Specifying a different number in the text box should automatically change the slider position of the corresponding track bar. The range of values for track bars and text boxes is from 0 to 100. The track bar should only be synchronized with the text box if the correct number (0 to 100) is entered in the text box.

7. The main form contains seven progress bars (the ProgressBar controls), the **Default** button, and the **Show** button. When the **Show** button is clicked, a second (non-modal) form with seven track bars (the TrackBar controls) appears. As you change the slider position of some track bar, the content of the corresponding progress bar should automatically change (the range of values for track bars and progress bars is from 0 to 100). When you click on one of the progress bars, the corresponding track bar is toggled between available and unavailable state. When you click the **Default** button, all progress bars and track bars return to their initial (zero) position; the accessibility of the track bars does not change.

8. The main form contains the **Show** button and seven text boxes with the text **text1** – **text7**. When the **Show** button is clicked, a second (non-modal) form appears with seven track bars (the TrackBar controls) and labels containing the same text as the text boxes in the main form. When you change the slider position of some track bar, the width of the corresponding text box in the main form should automatically change (the range of values for track bars is from 0 to 100). When changing the text in some text box, the text of the corresponding label in the second form should automatically change.

9. The main form contains the **Show** button, seven track bars (the TrackBar controls), and seven labels with the text **label1** – **label7**. The range of values for track bars is from 10 to 30, the initial value is 10. When the **Show** button is clicked, a second (non-modal) form appears with seven text boxes. The content of the text boxes must be synchronized with the text of the labels of the main

form. When you change the slider position of some track bar, the font size for the corresponding text box changes according to the position of the slider (in the range from 10 to 30). When you change the text in some text box, the text in the corresponding label should change accordingly.

10. The main form contains the **Show** button and seven text boxes with the text **text1** – **text7**. When the **Show** button is clicked, a second (non-modal) form appears with seven text boxes that must be synchronized with the corresponding text boxes of the main form. Synchronization must be performed whenever the text in any text box changes. The second form also contains the **Stop Synchronization** button. When the **Stop Synchronization** button is clicked, synchronization stops and the button name is changed to **Resume Synchronization**. Clicking the button again resumes synchronization mode, and the text specified in the text boxes of the *main form* is used for synchronization. When you close and reopen the second form, the state of the **Stop/Resume Synchronization** button does not change.

11. The main form contains the **Show** button and seven text boxes with the text **text1** – **text7**. When the **Show** button is clicked, a second (non-modal) form appears with seven labels; the text of label coincides with the text of the corresponding text box. When you click on any label, the background color of the corresponding text box toggles between white and gray. When you change the text in the text box with a white background, the text of the corresponding label is changed accordingly; when you change the text in the text box with a gray background, the label does not change. However, when you change the background color of some text box from gray to white, the text of the text box and the text of the corresponding label should be immediately synchronized.

12. The main form contains the **Show** button and seven text boxes with the text **text1** – **text7**. When the **Show** button is clicked, a second (non-modal) form appears with seven track bars (the TrackBar controls); slider position of each track bar coinsides with the length of the text in the corresponding text box. When editing text in some text box, the slider position of the corresponding track bar is automatically changed; when the slider position of some track bar is changed, the text in the corresponding text box is shortened or lengthened (text lengthening is performed by adding * characters). Track bars can take values from 0 to 25; text longer than 25 characters cannot be input into the text box.

25.5. DRAGDROP project: drag-and-drop mode

General guidelines. In all projects, the form must have a fixed size. Drag-and-drop mode has been discussed in detail in the ZOO project (Chapter 10); in addition, it was used in the LISTBOXES (Section 19.4) and HTOWERS (Section 24.3) projects. Working with menus was considered in the TEXTEDIT1 and TEXTEDIT2 projects (Sections 12.1, 13.1–13.3).

1. The form contains three multi-line text boxes. Implement the ability to drag and drop text files from **Explorer** onto one of the text boxes; as a result of a such action, the contents of the text file is added to the existing text of this text box. Only files with the .**txt** extension should be processed. In addition, implement the ability to drag and drop the non-empty content of one text box onto another (while holding down the Ctrl key); as a result of a such action, the contents of the source text box is added to the previous contents of the target text box. The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command removes text from the text box that has focus.

2. The form contains three multi-line text boxes and three labels (each label is placed above the corresponding text box). Initially, the labels contain the text **<No file>**. Implement the ability to drag and drop text files from **Explorer** onto one of the *labels* and automatically load the contents of these files into the appropriate text box. Only files with the .**txt** extension should be processed; the full file name must be displayed in the corresponding label. You can only drag and drop a file onto the label with the text **<No file>**. The form menu contains one **Command** submenu with the **Save**, **Clear**, and **Exit** menu items. The **Save** and **Clear** commands affect the focus text box; the **Save** command saves the new contents of the text box in the same file, the **Clear** command clears the text box along with the associated label (the label displays the text **<No file>** again).

3. The form contains four "usual" labels with the letters of nucleotides **A**, **C**, **T**, **G** and three "wide" labels that have a frame and contain gene sequences (in the beginning, wide labels are empty). The width of wide labels does not depend on the size of the text and is determined by the width of the form. Implement the ability to drag and drop a nucleotide letter from a usual label onto a wide label; the nucleotides are added to the end of the text of the wide label. Also, implement the ability to drag and drop the contents of one wide label onto another; this action adds all text of the source label to the end of the target label text. The form menu contains one **Command** submenu with the **Clear 1**, **Clear 2**, **Clear 3**, and **Exit** menu items. The **Clear** commands clear the wide label with the specified number. All **Clear** commands must use one common handler.

4. The form contains six labels with the text **label1** – **label6** and six text boxes with the text **textBox1** – **textBox6**. When you drag and drop a text box onto a label (while holding down the Ctrl key), the text and font of the label changes to the text and font of the text box (the position of the text box does not change). You can drag and drop the text box onto the label several times. The form menu contains one **Command** submenu with the **Bold**, **Italic**, and **Exit** menu items. The **Bold** and **Italic** menu items act as checkboxes to set or unset the bold and italic font mode for the text box that has focus.

5. The form contains six buttons with titles **button1** – **button6** and six empty list boxes (the ListBox controls). When you drag and drop a button onto the list box, the title of this button is inserted *to the specified position* of the list (the position of the button does not change). You can drag and drop a button onto the list several times. The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command clears the contents of the list with focus; if a *button* has focus, the command performs no action. To determine the number of an item in the list by the position of the mouse cursor, use the IndexFromPoint method (see Section 19.4).

6. The form contains six radio buttons with titles **color1** – **color6** (titles have different colors) and six rectangles with a white background (use Panel controls as rectangles). When you drag and drop a radio button onto a rectangle, the background color of the rectangle changes to match the color of the title of the radio button being dragged (the position of the radio button does not change). The form menu contains one **Command** submenu with **Color** and **Exit** menu items. The **Color** command shows the ColorDialog dialog box to change the color of the title of the *selected* radio button. Working with the ColorDialog control is described in Section 13.3.

7. The form contains a panel (the Panel control) and two buttons with titles **New** and **Trash**. Pressing the **New** button creates a new label located in a random place on the panel; the label text is a random capital Latin letter. Implement the ability to drag and drop the appeared labels onto new location on the panel. Dragging a label onto the **Trash** button removes the label. When you drag and drop one label onto another, the source label is removed and the text of the source label is *added* to the text of the target label. The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command removes all labels. All labels are also removed by clicking the **Trash** button. See the HTOWERS project (Section 24.1) for information on creating controls at runtime.

8. The form contains four panels (Panel controls) and a group of four radio buttons with titles **1** – **4** for selecting the current panel. There are three labels with text **label1** – **label3** on *each* panel in random places. Implement the ability to drag and drop labels from the *current* panel to any other (in addition, for the current panel, you can drag and drop labels within this panel). Labels located on other panels cannot be dragged. The form menu contains one **Command** submenu with the **Color** and **Exit** menu items. The **Color** command shows the ColorDialog dialog box to change the color of all labels in the current panel. See the HTOWERS project (Section 24.3) for information on dragging and dropping controls between group controls. Working with the ColorDialog control is described in Section 13.3.

9. Implement an application to test drag-and-drop mode. The form contains four multi-line text boxes, a label with the text **Label**, and a label with the text **String**. Any of the labels, as well as any objects from other programs, in particular from **Explorer**, can be dragged and dropped onto any empty text box. As a result, detailed information about the drag object is displayed in the text box (including the available formats, which can be determined using the GetFormats method). When you drag the **Label** label, the drag object is the label itself; when you drag the **String** label, the drag object is the string with its name (that is, the string object). The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command clears the text box that has focus.

10. The form contains a list box for adding file names by dragging and dropping them from **Explorer** (the full file name is added to the *end* of the list). The form also contains the **Trash** label. When dragging the list onto the **Trash** label (while holding down the Ctrl key), the current list item is deleted and the next list item (or the previous one if the *last* list item was deleted) becomes the current one; the position of the list box does not change. If the list box is empty, then dragging it is not allowed (that is, the dragging cursor has a prohibition sign). The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command clears the list box. See the LISTBOXES project (Section 19.4) for information on dragging and dropping list items.

25.6. TIMER project: timer-controlled programs

General guidelines. In all projects, the form must be resizable; when changing the form size, the size and position of the form controls must be adjusted accordingly (using the Anchor or Dock properties). Working with a timer was discussed in the CLOCK project (Chapter 7).

1. The form contains a combo box with a list of comments (the ComboBox control with DropDownStyle = **DropDown**; initially, it contains only one list item "–"), an empty list box with the **Results** label, a *stopwatch label* with the text **0:0**, and a button with the title **Start** used to start the stopwatch (when the stopwatch starts, the button title changes to **Stop**).

The stopwatch displays seconds and tenths of a second. When you stop the stopwatch, its text, along with the current comment from the combo box, is appended to the end of the **Results** list and the stopwatch label is set to **0:0**. To add a new comment to the list of comments, just input it in the text field of the combo box and press Enter. The form menu contains one **Command** submenu with the **Clear** and **Exit** menu items. The **Clear** command clears the **Results** list.

2. The form contains a label with the lowercase Latin letter **a** and two read-only text boxes with the labels **Time** and **Points scored**. In the upper part of the form there is a toolbar with four shortcut buttons **Start**, **10 sec**, **30 sec**, **60 sec**; the last three shortcut buttons form a group that necessarily contains one button

in the pressed state (initially, it is the button with the title **10 sec**). In addition, the form contains a list box with the **Top Scores** label, this list box displays the top 10 scores for the selected time mode (the time mode is determined by the shortcut buttons **10 sec**, **30 sec**, **60 sec**). List of top scores is sorted in descending order.

When you press the **Start** button, the countdown begins in the **Time** text box (in tenths of a second), the **Points scored** text box is reset to zero, and the Latin letter in the label changes. It is required to press the key with the specified Latin letter. When the key is pressed correctly, the counter in the **Points scored** text box increases by 1 and the letter in the label changes again. If the key is pressed incorrectly, the **Points scored** counter decreases by 1. The duration of one training session (in seconds) is determined by the selected time mode, that is, by the shortcut button in the pressed state.

After the completion of the training session, the **Top Scores** list is corrected, if necessary; if the new score is added into the list, then this is reported in the auxiliary dialog box (use the MessageBox.Show function). The lists of top scores for each mode are stored in the files **scores10.dat**, **scores30.dat**, **scores60.dat** and are read from them when changing the mode and when starting the program.

3. The form contains a label and two read-only text boxes with the labels **Time** and **Points scored**. In the upper part of the form there is a toolbar with five shortcut buttons: **Start**, “+”, “–”, “*”, “/”; the last four shortcut buttons form a group that necessarily contains one button in the pressed state (initially, it is the button with the title “+”). In addition, the form contains a panel with five radio buttons with empty labels and a list box with the **Top Scores** label, this list box displays the top 10 scores for the selected math operation mode (the math operation mode is determined by the shortcut buttons “+”, “–”, “*”, “/”). List of top scores is sorted in descending order.

When you press the **Start** button, the countdown begins in the **Time** text box (in tenths of a second), the **Points scored** text box is reset to zero, and a numerical expression with the selected math operation appears in the label (for example, **34 + 78 =**). The group of radio buttons displays 5 answer options. You need to click on the radio button with the correct option. If the answer is correct, the counter in the **Points scored** text box is increased by 1; if the answer is incorrect, it is decreased by 1. In any case, a new expression appears in the label. The duration of one training session is 30 seconds.

After the completion of the training session, the **Top Scores** list is corrected, if necessary; if the new score is added into the list, then this is reported in the auxiliary dialog box (use the MessageBox.Show function). The lists of the top scores for each mode are stored in the files **add.dat**, **sub.dat**, **mult.dat**, **div.dat** and are read from them when changing the mode and when starting the program.

For the “+” and “–” modes, the expressions must use numbers from 1 to 100, for the “*” mode, the expressions must use numbers from 1 to 10, for

the "/" mode, the first operand must be two-digit number, the second one-digit number, and the result must be an integer.

4. The form contains a label with text displaying the current system time of the computer in the **hh:mm** format and three checkboxes associated with a separate alarm clock. The text near the checkbox indicates the alarm time and is also in the **hh: mm** format. Alarm clock is activated at the specified alarm time if the corresponding checkbox is checked; in this case, the checkbox changes state to **Indeterminate** and the sound signal (from the .**wav** sound file) is played for 10 seconds. If the sound file is less than 10 seconds long, the file is played in a loop (use the System.Media.SoundPlayer class to play .**wav** file).

To turn off the signal early, just uncheck the corresponding checkbox. After 10 seconds of sound signal playback, the checkbox is automatically unchecked. When you check any unchecked checkbox, a dialog box with two drop-down lists is displayed, in which you can set a new alarm time (hours and minutes); the default alarm time is the time previously associated with that alarm.

When the program finishes, it saves each alarm time and its current state in the **alarm.dat** text file. The saved data is restored when the program is started.

5. The form contains a read-only text box with the text **0:0** and the label **Time**, an empty list box with the label **Results**, and a panel that contains 6 small square labels numbered **1** – **6**. The form menu contains the **Command** submenu with the **Start** and **Exit** menu items.

When the **Start** command is executed, all the panel labels change their location on the panel randomly and are filled with a red backgroupd color, and the time count begins in the **Time** text box (in tenths of a second). You need to quickly click on all the panel labels in the ascending order of their numbers. Clicking on the correct label makes its background green. As soon as all 6 labels are clicked, the time count stops. If the **Time** text box contains a value less than **10:0** (that is, less than 10 seconds), then the message box with the text **You win!** is displayed and this time value is added to the top of the **Results** list box. Otherwise, the message box **You lost** is displayed.

The **Start** menu item should be available only when the game is stopped. The list of results must be stored in the **results.dat** file and read from this file each time the program is started. When changing the position of labels on the panel, it is necessary that labels do not intersect (see Comment 2 in Section 4.3).

6. The form contains a panel and two read-only text boxes with the labels **Time** and **Points scored**. In addition, the form contains a list box with the label **Top 5 scores**. Initially, the **Time** text box contains the number **30**; the **Points scored** text box contains the number **0**. The form menu contains one **Command** submenu with the **Start** and **Exit** menu items.

When the **Start** command is executed, the countdown begins in the **Time** text box (from 30 to 0, in seconds) and a small label with the number **50** appears

on the panel in a random place (the label size is 10 × 10 pixels). The number on the label decreases by 1 every tenth of a second. When you click on the label, the number in the **Points scored** text box increases by the number on the label (or decreases if the number on the label is negative) and the label is displayed elsewhere on the panel (again with the number **50**). When you click outside the label, the number **10** is subtracted from the **Points scored** value. After 30 seconds, the game ends. If the **Points scored** text box contains a positive number, then the message box with the text **You win!** is displayed, otherwise the message **You lost** is displayed; the list of the top scores is adjusted if necessary.

The **Start** menu item should be available only when the game is stopped. The top score list must be stored in the **score.dat** text file and read from this file each time the program is started.

7. The form contains a panel, a button with the title **Start**, three read-only text boxes with the labels **Accuracy**, **Time**, and **Result** (initially, the text boxes contain zeros), and a list box with the label **5 best results**.

When you click on the **Start** button, its title changes to **Stop**, the time count begins in the **Time** text box (in tenths of a second), and, in one of the corners of the panel, a square framed label without a text is displayed (the panel corner is selected randomly). It is required to drag this label with the mouse exactly to the center of the panel and press the **Stop** button (when dragging, the label must follow the mouse cursor). After that, the **Accuracy** text box displays two numbers: the horizontal and vertical deviations from the correct position (in pixels) and the **Result** text box displays a number calculated as follows: 1 is added to the time obtained (in seconds, with one fractional digit) and this number is multiplied by the sum of the absolute values of the deviations. See the MOUSE project (Section 9.1) for information on dragging with the mouse.

If the number in the **Result** text box is less than **50**, then the message box with the text **You win!** is displayed, otherwise the message **You lost** is displayed; the list of the best results is adjusted if necessary. The list of the best results must be stored in the **results.dat** text file and read from this file each time the program is started.

8. The form contains two read-only text boxes with the labels **Missiles** and **Time to explosion**, a panel, and a single-character label depicting an airplane (the **Wingdings** font, symbol **Q**) located in the left top corner of the panel. The mouse cursor on the panel looks like a cross. The form menu contains the **Command** submenu with the **Start** and **Exit** menu items.

When the **Start** command is executed, the airplane's label begins a straight-line movement on the panel (the increments of the `Left` and `Top` properties should be in the range 1–3; they are determined randomly before starting the game and are performed every tenth of a second) and the number **4** appears in the **Missiles** text box. Clicking on the panel marks the point of launching a missile, which

will explode after 2 seconds (in the **Time to explosion** text box, the countdown begins from **2.0** to **0.0**, in tenths of a second); the missile launch point on the screen should be marked with a red-colored label of the size 2 × 2 pixels. At the moment of the explosion, the size of the red-colored label increases to 30 × 30 pixels (the destruction area).

If, at the moment of the explosion, the airplane is in the destruction area, then the message box appears with the text **The airplane is shot down** and the game ends. Otherwise, nothing happens. You cannot launch a new missile before the explosion of a previously launched one. If all the missiles have been used without results or the airplane has reached the border of the panel, the message **You lost** is displayed. Use the IntersectsWith method to verify that the airplane is in the destruction area (see Comment 2 in Section 4.3).

The **Start** menu item should be available only when the game is stopped.

25.7. REGISTRY project: dialog boxes and working with the Windows registry

General guidelines. Working with the Windows registry is described in the IMGVIEW project (Sections 21.5–21.6). Working with the OpenFileDialog dialog box is described in the TEXTEDIT1 project (Section 12.3). To organize dialogs related to choosing a font and color, you should use the FontDialog and ColorDialog controls; an example of working with these controls is given in the TEXTEDIT2 project (Sections 13.3–13.4). Working with the SplitContainer control is described in the IMGVIEW project (Sections 21.1–21.2). Working with selections in the text boxes is discussed in the TEXTBOXES project (Section 8.1).

1. The form contains the SplitContainer control with a vertical splitter orientation. The left and right panels of the SplitContainer control contain one label and one multi-line text box. Initially, focus is on the left text box; pressing the Tab key toggles focus between the TextBox controls. When the form is resized, the panels of the SplitContainer control are sized proportionally (you cannot change the width of the panels by dragging the splitter; both panels always have the same width). The form menu contains one **File** submenu with the **Open...**, **Save**, **Compare**, and **Exit** menu items.

The **Open...** command displays the OpenFileDialog dialog box, which allows you to open existing text files, as well as create new ones (if the required file is missing, it is created automatically). When the required file is open, its text is loaded into the active text box and the full file name is displayed in the label above this text box.

The **Compare** command is available only if both TextBox controls contain loaded data; it compares the contents of the left and right text boxes and positions the cursor before the first differing character in the *left* text box (pressing

Tab should set the cursor before the first differing character in the *right* text box).

If the text of the text box is changed by the user, the symbol "*" is indicated in the label before the file name. The **Save** command saves the contents of the active text box in the file with the same name; after that, the symbol "*" disappears in the label.

At the end of the program, the file names are saved in the Windows registry; the next time the program is started, they are read from the registry; the size and position of the form and the position of the cursor in each text box should also be restored. When you try to close the program without saving the changed contents of the files, a standard dialog box appears asking if you want to save the changed file; the options are **Yes**, **No**, **Cancel**. If there are two unsaved files, two dialog boxes are displayed sequentially (unless you selected **Cancel** in the first dialog box).

2. The form contains a multi-line text box (the size of the text box is automatically resized when the form is resized), a drop-down list of options of number system conversion: **10 => 2** (this option is selected by default), **10 => 16**, **2 => 10**, **16 => 10**, **16 => 2**, **2 => 16**, and the **Convert** button. After the first launch of the program, at the beginning of its work, the text box is not available for editing. The form menu contains one **File** submenu with the **Open...**, **Save**, and **Exit** menu items.

The **Open...** command displays the OpenFileDialog dialog box, which allows you to open existing text files. As a result, the text of this file is loaded into the text box and the name of the loaded file is displayed in the form title bar.

The **Convert** button converts the number selected by the user in the text box from one number system to another (the selected number is replaced by the converted number; the converted number remains selected). If nothing is selected or the selection contains invalid data, then nothing happens.

If the text of the loaded file has been changed, the symbol "*" is displayed in the form title bar after the file name. The **Save** command saves the contents of the text box in the file with the same name; after that, the symbol "*" disappears in the form title bar.

At the end of the program, the name of the currently open file is saved in the Windows registry; the next time the program is started, it is read from the registry; the last conversion option, the size and position of the form, and the position of the cursor in the text box are also restored. When you try to close the program without saving the changed file contents, a standard dialog box appears asking if you want to save the changed file; the options are **Yes**, **No**, **Cancel**.

3. The form contains a list box with the **Playlist** label, the buttons **Play/Stop**, **Up** and **Down**, and the NumericUpDown control with the **Duration (sec)** label. The list box is automatically resized when the form is resized. If the

list box is empty, the buttons are inactive. The form menu contains one **Command** submenu with the **Add file**, **Add folder**, **Clear**, and **Exit** menu items.

The **Add file** and **Add folder** commands show the OpenFileDialog dialog box, which allows you to open existing wav-files; the **Add file** command adds the selected file to the list, the **Add folder** command adds *all* wav-files from the folder to the list (files are added to the end of the list). The last item added to the list becomes the current item. After adding at least one item to the list, the **Play/Stop** button becomes available; if the list contains more than one item, the **Up** and **Down** buttons become available.

Clicking the **Play/Stop** button starts playback of the current file from the list or stops playback of a file. The **Duration (sec)** counter allows you to specify the playback time of each file (in seconds); the default time is **10** seconds (if the file duration is less than the specified time, the file is played cyclically). The **Up** and **Down** buttons allow you to move the current item in the list up or down. When the playing time ends, the file located in the list after the current one starts playing automatically (this item of the list becomes the current one). Files are played cyclically. During playback, the ListBox control, the NumericUpDown control, and the **Up** and **Down** buttons are disabled. Use the System.Media.SoundPlayer class to play wav-files; working with list box items is described in the LISTBOXES project (Sections 19.2–19.3).

At the end of the program, the playlist, the position of the current list item, the playback time, the size and position of the form are saved in the Windows registry. The next time the program is started, it should restore the saved state.

4. The form contains the NumericUpDown control with the **Symbol code** label and a panel (the GroupBox control) with a label containing one character of 48 points size (the code of this symbol is specified in the NumericUpDown control). Initially, the program is configured for the **Wingdings** font; the name of the font is specified in the title of the GroupBox control. The form menu contains one **Command** submenu with the **Font...** and **Exit** menu items.

The **Font...** command shows the FontDialog dialog box, which allows you to change the name and style of the font, but not its size. The font size should be changed automatically when the form is resized; the initial form size cannot be reduced. When value of the NumericUpDown control changed, the corresponding symbol for the selected font is displayed in the panel. The displayed symbol must be centered vertically and horizontally relative to the border of the GroupBox control; to do this, set the appropriate values to the label properties AutoSize, Dock, TextAlign.

At the end of the program, the value of the NumericUpDown control, the name and style of the current font, as well as the size and position of the form are saved in the Windows registry. The next time the program is started, it should restore the saved state.

5. The form contains the NumericUpDown control with the **KnownColor number** label and a panel (the GroupBox control). When the form is resized, the panel size changes proportionally. When you input the number of one of the standard named colors from the KnownColor enumeration into the NumericUpDown control, the GroupBox panel is filled with this color and the name of this color is displayed in the title of this panel. The range of valid values for the NumericUpDown control must match the range of all standard named colors of the KnownColor enumeration: from **AliceBlue** to **YellowGreen**. The form menu contains one **Command** submenu with the **Color...** and **Exit** menu items.

The **Color...** command shows the ColorDialog dialog box, which allows you to select a color for the background of the GroupBox panel. If the selected color is one of the standard named colors, then its number appears in the NumericUpDown control, if the selected color is not a standard named color, then an error message is displayed in the standard message box and the panel background does not change. Working with colors is described in the COLORS and LISTBOXES projects (Chapter 18 and Section 19.1).

At the end of the program, the value of the NumericUpDown control and the size and position of the form are saved in the Windows registry. The next time the program is started, it should restore the saved state.

6. The form contains an empty multi-line text box (the TextBox control) unavailable for editing, with a gray background, the **Open** button, and 3 track bars (the TrackBar controls). Each track bar can take 10 values: from 0 to 9. The track bars are oriented vertically and are located in left-hand side of the form. A label is displayed above each track bar that contains the current value of that track bar (a number from **0** to **9**). The **Open** button is located under the track bars, the multi-line text box occupies the rest (right-hand) part of the form along its entire height.

When the form is resized, the width and height of the multi-line text box, as well as the height of the track bars, must change; the minimum allowable size of the form must be adjusted so that they provide the display of all its controls.

When you set the correct three-digit *lock code* with the track bars and then click the **Open** button (or press Enter), the content of the **notebook.txt** file (if this file exists) is loaded into the TextBox control, the background of the TextBox control turns white, the text box becomes editable, and the button title changes to **Close** (if the code is set incorrectly, the appearance of the TextBox control and the button does not change).

The correct lock code is stored in the Windows registry; if there is no the corresponding subkey in the registry, then the code is **000**. When the text box is editable, you can set new lock code using the track bars.

When you click the **Close** button, the text is saved in the **notebook.txt** file, the code is saved in the Windows registry, the text box is cleared, its background

is grayed out, the button title is changed to **Open**, and the values of track bars are changed randomly.

At the end of the program, the size and position of the form are additionally saved in the Windows registry and are restored the next time the program is started.

7. The form contains the SplitContainer control with a vertical splitter orientation. The list box is located on the left panel of the SplitContainer control, a multi-line text box is located on the right panel. The list box and text box are automatically resized when the form is resized; when the form width changes, the width of the *text box* changes. The splitter between the left and right panel of the SplitContainer control can be dragged. The Tab key allows you to toggle focus between the list box and the text box. The form menu contains one **File** submenu with the **Open...**, **Close**, and **Exit** menu items.

The **Open...** command displays the OpenFileDialog dialog box which allows you to open existing text files, as well as create new ones (if the required file is missing, it is created automatically). When the required file is open, its full name is added to the end of the list box (and becomes the selected list item) and its text is loaded into the text box. In the future, to load this file into the text box, it is enough to select its name in the list box. If, when executing the **Open...** command, the name of the file is already included in the list box, then this name in the list is made selected. When a list item loses selection, its associated text is automatically saved in the appropriate file.

The **Close** command removes the name of the selected file from the list of files; this action also automatically saves the text in the file. When a list item is deleted, the next item is selected; if the next item is absent, the previous item is selected. If the list box is empty, then the text box is not editable and the **Close** menu item is unavailable. Working with list box items is described in the LISTBOXES project (Sections 19.2–19.3).

At the end of the program, the file list is saved in the Windows registry. The index of the selected list item, the position of the cursor in the text box, the size and position of the form, the position of the splitter between the left and right panels are also saved in the registry. The next time the program is started, it should restore the saved state.

25.8. MDIFORMS project: MDI applications

Working with MDI applications is described in the JPEGVIEW project (Chapter 22).

1. The MDI main form initially contains one special child form with the title **Clipboard**, which is entirely occupied by the multi-line TextBox control. This child form acts as the application's own *clipboard*. The MDI application menu includes three submenus: **File** (the **Open** and **Exit** menu items), **Clipboard** (the

Cut, **Copy**, **Paste** menu items), and **Window** (the **HTile**, **VTile**, **Cascade**, **Arrange Icons** menu items, as well as a list of child forms). The commands of the **Window** submenu provide standard MDI application actions related to the placement of child forms and their selection (see Section 22.2).

The **Open** command allows you to create or load a text file and display it in a new child form with the TextBox control (in this case, the **Close** menu item appears in the **File** submenu; this command closes the active child form). In the OpenFileDialog control used to select the file name, you should set the file mask (the Filter property) to display only text files (with the .**txt** extension). If a file with the specified name does not exist, then it is created. The full name of the created or loaded file is displayed in the title bar of the corresponding child form; when you try to reload an existing file, a new child form is not created; instead, the child form that already contains the specified file becomes active. When the child form is closed, the corresponding text file is automatically saved.

When executing the commands **Cut**, **Copy**, **Paste**, the **TextBox** control of the **Clipboard** child form should be used (instead of the standard Windows clipboard): the text cut or copied from any other child form should be placed on the **Clipboard** child form (its previous content is deleted). When the **Paste** command is executed, the text from the **Clipboard** form should be inserted into the current position of the active child form. The contents of the **Clipboard** form can be edited, however, commands related to copying, cutting and pasting cannot be executed for it; furthermore, this child form cannot be closed.

2. The MDI main form initially contains no child forms. The MDI application menu includes two submenus: **File** (the **Open** and **Exit** menu items) and **Group** (the **Open Group** menu item).

The **Open** command allows you to create or load a text file and display it in a new child form with the TextBox control; the **Open Group** command allows you to immediately load all text files from the selected directory. In the OpenFileDialog control used to select the file name, you should set the file mask (the Filter property) to display only text files (with the .**txt** extension). The OpenFileDialog control is also used for group file loading; when the **Open Group** command is executed, it is sufficient to select *one* of the text files in the required directory to load *all* text files from this directory. When executing the **Open** command, you can specify the name of a non-existent file; in this case, it is created. The full name of the created or loaded file is displayed in the title bar of the corresponding child form; when you try to reload an existing file, a new child form is not created; instead, the child form that already contains the specified file becomes active. A similar condition must be satisfied for a group loading: already loaded files are not reloaded.

If there is at least one child form, the **Close** menu item appears in the **File** submenu (this command closes the active child form) and the **Close Group** and

Close All menu items appear in the **Group** submenu (the **Close Group** command closes the active child form and all other child forms with files from the same directory as the active child form file; the **Close All** command closes all child forms). When the child form is closed, the corresponding text file is automatically saved.

In addition, if there is at least one child form, the **Window** submenu appears in the application menu with the **HTile**, **VTile**, **Cascade**, **Arrange Icons** menu items, as well as with a list of child forms. The commands of the **Window** submenu provide standard MDI application actions related to the placement of child forms and their selection (see Section 22.2).

3. The MDI main form initially contains no child forms. The MDI application menu includes two submenus: **File** (the **Open** and **Exit** menu items) and **Actions** (the **Shift Forward**, **Shift Backward**, and **Union** menu items; these menu items are available only if there are at least *two* child forms).

The **Open** command allows you to create or load a text file and display it in a new child form with a TextBox control. In the OpenFileDialog control used to select the file name, you should set the file mask (the Filter property) to display only text files (with the .**txt** extension). When executing the **Open** command, you can specify the name of a non-existent file; in this case, it is created. The full name of the created or loaded file is displayed in the title bar of the corresponding child form; when you try to reload an existing file, a new child form is not created; instead, the child form that already contains the specified file becomes active. If there is at least one child form, the **File** submenu displays the **Close** menu item that closes the active child form, and the **Close All** menu item that closes all child forms. When the child form is closed, the corresponding text file is automatically saved.

The **Shift Forward**, **Shift Backward**, and **Union** commands change the contents of the child forms as follows. The **Shift Forward** command performs a *cyclic shift forward*, that is, it moves the contents of the first child form to the second child form, the contents of the second child form to the third child form, …, the contents of the last child form to the first child form. The **Shift Backward** command performs a *cyclic shift backward*, that is, it moves the contents of the second child form to the first child form, the contents of the third child form to the second child form, …, the contents of the first child form to the last child form. The **Union** command combines the contents of all child forms into the active child form; the text is added in the order of child form numbers, starting with the active child form (for example, when executing the **Union** command for the third of five loaded forms, the initial text of the child forms with the following numbers will be written in the third form: 3, 4, 5, 1, 2).

If there is at least one child form, the **Window** submenu appears in the application menu with the **HTile**, **VTile**, **Cascade**, **Arrange Icons** menu items, as well as with a list of child forms. The commands of the **Window** submenu pro-

vide standard MDI application actions related to the placement of child forms and their selection (see Section 22.2).

4. The MDI main form initially contains no child forms. The MDI application menu includes two submenus: **File** (the **Open** and **Exit** menu items) and **Actions** (the **Move**, **Add**, and **Swap** menu items; these menu items are available only if there are at least *two* child forms).

The **Open** command allows you to create or load a text file and display it in a new child form with a TextBox control. In the OpenFileDialog control used to select the file name, you should set the file mask (the Filter property) to display only text files (with the .**txt** extension). When executing the **Open** command, you can specify the name of a non-existent file; in this case, it is created. The full name of the created or loaded file is displayed in the title bar of the corresponding child form; when you try to reload an existing file, a new child form is not created; instead, the child form that already contains the specified file becomes active. If there is at least one child form, the **File** submenu displays the **Close** menu item that closes the active child form, and the **Close All** menu item that closes all child forms. When the child form is closed, the corresponding text file is automatically saved.

The **Move** command changes the order of the child forms by moving the active form to the end of the list of child forms. To implement this command, it is enough to close the active child form (and save, if necessary, its contents in the corresponding file) and create a new child form with the same contents.

The **Add** and **Swap** commands modify the contents of the child forms. The **Add** command adds the contents of the form that *follows* the active form to the contents of the active form. The **Swap** command swaps the contents of the active child form and the form that follows the active form. The first child form is assumed to follow the last child form.

If there is at least one child form, the Window submenu appears in the application menu with the **HTile**, **VTile**, **Cascade**, **Arrange Icons** menu items, as well as with a list of child forms. The commands of the **Window** submenu provide standard MDI application actions related to the placement of child forms and their selection (see Section 22.2).

References

1. *Abramyan M. E.* Visual C# by examples. – SPb: BHV-Petersburg, 2008. – 482 p. (In Russian.)

2. *Abramyan M. E.* .NET Framework: The main types of the standard library. Working with arrays, strings, files. Objects, interfaces, generics. LINQ technology. – Rostov-on-Don: SFedU Press, 2014. – 218 p. (In Russian.)

3. *Albahari J., Albahari B.* C# 6.0 in a Nutshell. The definitive reference. 6th Edition. – Boston: O'Reilly, 2016. – 1114 p.

4. *Graham R., Knuth D., Patashnik O.* Concrete mathematics. A foundation for computer science. Reading: Addison-Wesley, 1989. – 625 p.

Educational edition

Abramyan Mikhail Eduardovich

User interface development based on Windows Forms class library

Computer typesetting by *M. E. Abramyan*

Подписано в печать 29.06.2021 г.
Бумага офсетная. Формат 60×84 1/16. Тираж 30 экз.
Усл. печ. лист. 16,16. Уч. изд. л. 10,5. Заказ № 8073.

Издательство Южного федерального университета.

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1, тел (863) 243-41-66.


9 785927 538300