

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ
до виконання самостійних робіт з дисципліни
«Основи програмної інженерії»
для студентів спеціальності
121 «Інженерія програмного забезпечення»
усіх форм навчання

2020

Методичні вказівки до виконання самостійних робіт з дисципліни «Основи програмної інженерії» для студентів спеціальності 121 «Інженерія програмного забезпечення» усіх форм навчання /Уклад.: Каплієнко Т.І., Качан О.І., Федорченко Є.М. – Запоріжжя: НУ «Запорізька політехніка», 2020. – 98 с.

Укладачі: Т.І. Каплієнко, к.т.н., доцент кафедри ПЗ,
О.І. Качан, старший викладач кафедри ПЗ,
Є.М. Федорченко, старший викладач кафедри ПЗ.

Рецензент: А.О. Олійник, к.т.н., доцент кафедри ПЗ.

Відповідальний
за випуск: С.О. Субботін, зав. каф. ПЗ, д.т.н., професор.

Затверджено
на засіданні кафедри
"Програмні засоби"

Протокол № 1 від 18.08.2020 р.

ЗМІСТ

1 ПІДХОДИ ТА МЕТОДИ У РОЗРОБЦІ ТЕКСТОВОГО РЕДАКТОРУ НА МОВІ C#	4
1.1 Короткі теоретичні відомості	4
1.2 Завдання до роботи.....	35
1.3 Хід виконання роботи.....	35
1.4 Контрольні запитання	35
2 РОБОТА З ДРУКОМ ТА СТВОРЕННЯ ЕЛЕМЕНТІВ КЕРУВАННЯ У АЛГОРИТМІЧНІЙ МОВІ C#	36
2.1 Друк.....	36
2.2 Створення елементів керування за допомогою .NET Framework.....	55
2.3 Завдання до роботи.....	78
2.4 Хід виконання роботи.....	78
2.5 Контрольні запитання	79
3 РОБОТА З МЕНЮ ТА ОБРОБКА ВИНЯТКІВ У C#.....	80
3.1 Короткі теоретичні відомості	80
3.2 Завдання до роботи.....	95
3.3 Хід виконання роботи.....	95
3.4 Контрольні запитання	95
4 ЗАВДАННЯ ДЛЯ САМОСТІЙНИХ РОБІТ	96
4.1 Завдання до роботи.....	96
4.2 Хід виконання роботи.....	96
4.3 Зміст звіту.....	97
ПЕРЕЛІК ПОСИЛАНЬ	98

1 ПІДХОДИ ТА МЕТОДИ У РОЗРОБЦІ ТЕКСТОВОГО РЕДАКТОРУ НА МОВІ C#

1.1 Короткі теоретичні відомості

Елементи керування. Компоненти, що забезпечують взаємодію між користувачем і програмою. Серед Visual Studio.NET надає велику кількість елементів, які можна згрупувати у декількох функціональних групах.

Група меню. Багато користувачів налаштовують інтерфейс застосунків на свій смак: одним подобається наявність певних панелей інструментів, іншим – індивідуальне розташування вікон. Але в будь-якому застосунку буде присутній меню, що містить у собі доступ до усіх можливостей та налаштувань програми. Елементи **MainMenu**, **ContextMenu** представляють собою готові форми для внесення заголовків і пунктів меню.

Діалогові вікна. Виконуючи різні операції з документом – відкриття, збереження, друк, попередній перегляд – виконується взаємодія з відповідними діалоговими вікнами. Розробникам .NET не доводиться займатися створенням вікон стандартних процедур: елементи **OpenFileDialog**, **SaveFile Dialog**, **ColorDialog**, **PrintDialog** містять вже готові операції.

Елементи керування **Button**, **LinkLabel**, **ToolBar** реагують на натискання кнопки миші та негайно запускають будь-яку дію.

Більшість застосунків надають можливість користувачу уввісти текст та у свою чергу виводять різну інформацію у вигляді текстових записів. Елементи **TextBox**, **RichTextBox** приймають текст, а елементи **Label**, **StatusBar** виводять її. Для обробки введеного користувачем тексту, як правило, слід натиснути на один або декілька елементів із групи командних об'єктів.

Група перемикачів. Застосунок може містити кілька визначених варіантів виконання дії або завдання. Елементи керування цієї групи надають можливість обрання користувачеві. Це одна з найбільш великих груп елементів до якої належать **ComboBox**, **ListBox**, **ListView**, **TreeView**, **NumericUpDown** та багато інших.

Група контейнерів. Як правило, елементи цієї групи розташовано на формі та призначено підкладкою кнопкам, текстовим полям, спискам – тому вони і називаються контейнерами. Елементи


Panel, GroupBox, TabControl, крім усього іншого, поділяють можливості програми на логічні групи, забезпечуючи зручність роботи.

Група графічних елементів. Навіть самий простий застосунок Windows містить графічні іконки, заставку, вбудовані зображення. Для розташування та відображення їх на формі використовуються елементи для роботи з графікою – **ImageList, PictureBox**.

1.1.1 Створення головного меню

Більшість Windows-застосунків оснащено головним меню, яке представляє собою ієрархічну структуру виконуваних функцій та команд. Практично всі функції, які можна здійснити за допомогою елементів керування, мають свій аналог у вигляді пункту меню. Для створення головного меню використовується елемент керування **MainMenu**, який розташовано на панелі інструментів **ToolBox**. Створюється новий застосунок та отримує назву **NotepadCSharp**. Встановлюються належні властивості форми. Приклади створення головного меню наведено у таблиці 1.1.

Таблиця 1.1 – Створення головного меню

Form1, форма, властивість	Значення
Name	frmmain
Icon	 Code\Glava2\NotepadCSharp\Icon\README.ICO
Text	Notepad C#
WindowState	Maximized

Надалі належить заповнити рядки меню наступними пунктами (рис. 1.1).

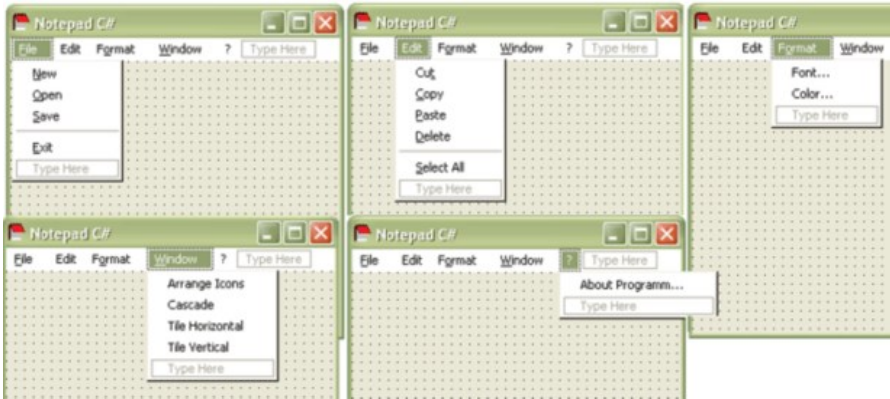


Рисунок 1.1 – Пункти головного меню програми Notepad C#

Кожен пункт головного меню має своє вікно властивостей, у якому подібно до інших елементів керування задаються значення властивостей **Name** та **Text** (рис. 1.2). У полі **Text** перед словом **New** стоїть знак **&** – так званий «амперсанд», який вказує, що **N** повинно бути підкреслено і буде частиною вбудованого клавіатурного інтерфейсу Windows. Коли користувач на клавіатурі натискає клавішу **Ctrl** і потім **N**, виводиться підменю **New**.

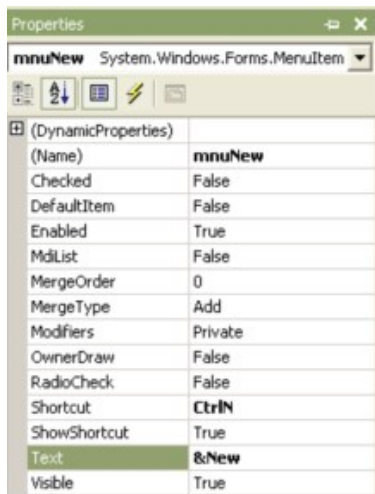


Рисунок 1.2 – Властивості пункту меню **New**

У Windows є ще інтерфейс для роботи із так званими швидкими клавішами або акселераторами. Поєднання клавіш вказують з перерахування **Shortcut**. Слід призначати стандартним пунктам загальноприйнятій поєднання клавіш. Горизонтальна розділова лінія використовується в тих випадках, коли треба візуально відокремити подібні групи завдань, – для її появи у властивості **Text** пункту меню вводиться знак **-**. Для використання пунктів меню в коді, їм також призначають імена (властивість **Name**), які особливо важливі, тому що пунктів меню звичайно буває багато. Властивості пунктів меню в застосунку Notepad C# наведено у таблиці 1.2.

Таблиця 1.2. – Пункти головного меню програми Notepad C#

ame	Text	Shortcut
mnuFile	&File	
mnuNew	&New	CtrlN
mnuOpen	&Open	CtrlO
mnuSave	&Save	CtrlS
menuItem5	-	
mnuExit	&Exit	AltF4
mnuEdit	&Edit	
mnuCut	Cu&t	CtrlX
mnuCopy	&Copy	CtrlC
mnuPaste	&Paste	CtrlV
mnuDelete	&Delete	Del
mnuSelectAll	&SelectAll	CtrlA
mnuFormat	F&ormat	
mnuFont	Font...	
mnuColor	Color...	
mnuWindow	&Window	
mnuArrangeIcons	Arrange Icons	
mnuCascade	Cascade	
mnuTileHorizontal	Tile Horizontal	
mnuTileVertical	Tile Vertical	
mnuHelp	?	
mnuAbout	About Programm...	

Рекомендовано самостійно обирати для відповідних пунктів меню сполучення клавіш, які не вказано в таблиці 1.2.

1.1.2 Створення MDI-застосунків

Такі програми, як Notepad та Microsoft Paint відносяться до SDI (Single – Document Interface) застосунків, здатних працювати тільки із одним документом. Інші, такі як Microsoft Word або Adobe Photoshop підтримують роботу відразу з декількома документами та мають назву MDI-застосунків (Multiple – Document Interface) (рис. 1.3 та 1.4).

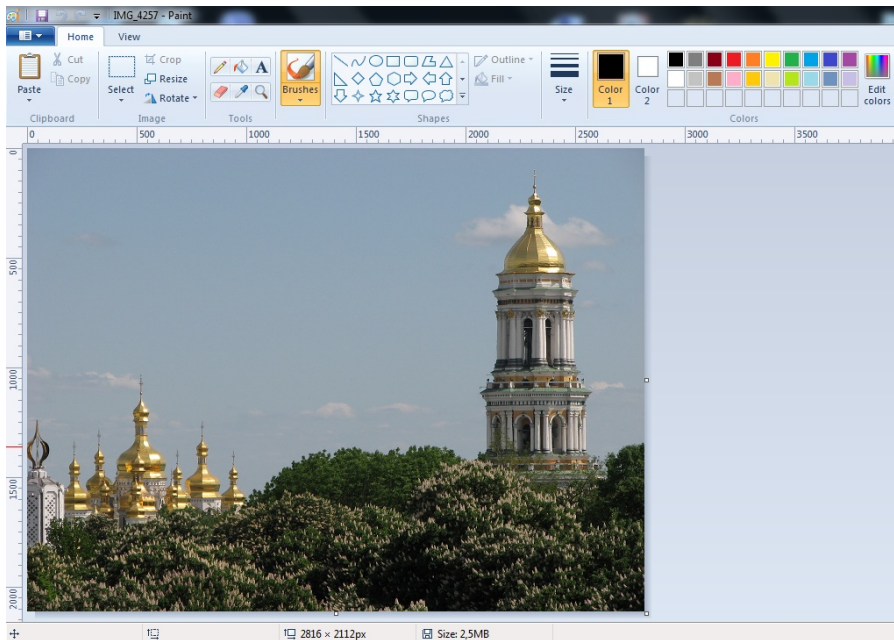


Рисунок 1.3 – SDI-застосунок Microsoft Paint

У MDI-застосунках головна форма містить у собі кілька документів, кожен з яких є полотном у графічних програмах або полями для тексту в редакторах.

У вікні Solution Explorer натисканням правою кнопкою миші на імені проекту в контекстному меню обирається Add Windows Form Надалі у вікні вводиться назва форми – **blank.cs**. У поточному проєкті було створено нову форму, яка має назву дочірньої. Надалі у

режимі дизайну необхідно перетягнути на неї елемент керування **RichTextBox**. На відміну від елементу **textBox**, розмір вмісту тексту в ньому не обмежується 64Кб. Крім того, **RichTextBox** дозволяє редагувати колір тексту, додавати зображення. Властивість **Dock** цього елементу встановлюється на значення **Fill** (рис. 1.5).



Рисунок 1.4 – MDI-застосунок Adobe Photoshop

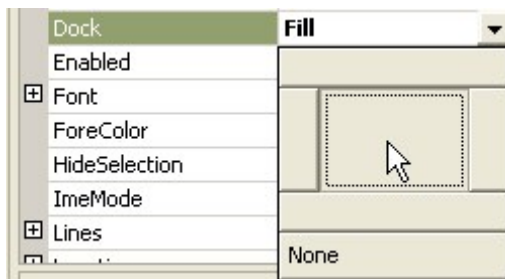


Рисунок 1.5 – Властивість **Dock** елементу **RichTextBox**

Надалі у режимі дизайну форми **Frmmain** встановлюється властивість **IsMdiContainer** у значення **true**. Колір форми при цьому стає темно-сірим. Нові документи тепер будуть з'являтися при натисканні пункту меню **New** (на який встановлено поєднання клавіш **Ctrl+N**), тому двійним натисканням у цьому пункті виконується перехід до редагування коду:

```
private void mnuNew_Click(object sender, System.EventArgs e) {
    blank frm = new blank();
    frm.MdiParent = this;
    frm.Show();
}
```

У режимі виконуваної програми при натисканні клавіш **Ctrl+N** або вибору пункту меню **New** з'являється кілька вікон, розташованих каскадом. Однак заголовок у них всіх однаковий – **blank**. При створенні декількох документів, наприклад, у Microsoft Word, вони називаються **Документ N**, де **N** – номер документу. Якщо перейти до редагування коду форми **blank**, то в класі **blank** необхідно задекларувати змінну **DocName**:

```
public string DocName = "";
```

Надалі необхідно перейти до коду форми **Frmmain** та в класі **Frmmain** оголосити змінну **openDocuments**:

```
private int openDocuments = 0;
```

Необхідно надати змінній **DocName** частину назви за шаблоном, до якого додано лічильник кількості відкритих документів, потім це значення треба надіслати властивості **Text** створюваної форми **frm**:

```
private void mnuNew_Click(object sender, System.EventArgs e) {
    blank frm = new blank();
    frm.DocName = "Untitled" + ++openDocuments;
    frm.Text = frm.DocName;
    frm.MdiParent = this;
    frm.Show();
}
```

Якщо запустити відкомпільовану програму на виконання, то можна отримати результат перетворення, який створить для усіх нових документів різні заголовки (рис. 1.6).

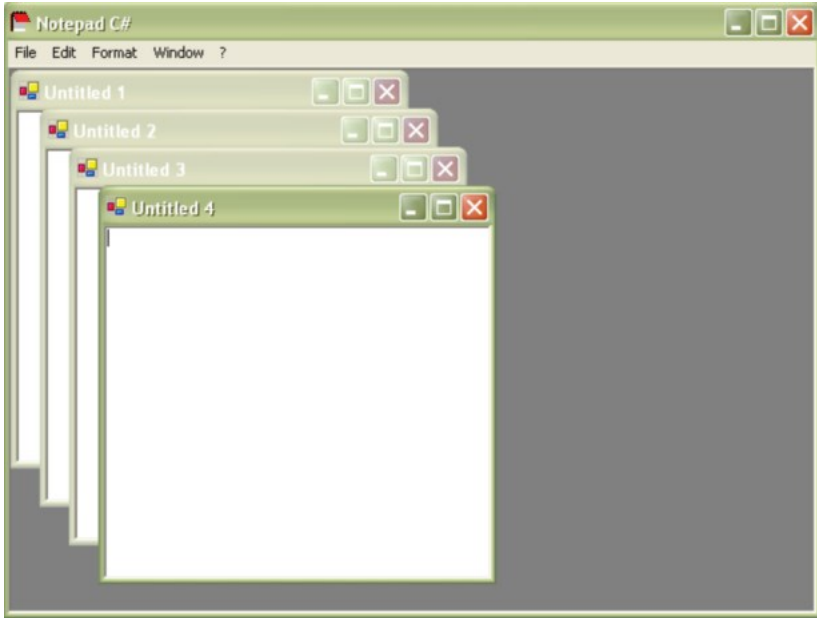


Рисунок 1.6 – Нові документи отримують впорядковані назви

1.1.3 Перелічення MdiLayout

При роботі з декількома документами у MDI-застосунках зручно впорядковувати вигляд цих документів на екрані. Можливо розподілити форми вручну, але при роботі з великою кількістю документів це є трудомісткім. Тому в пункті меню Window необхідно реалізувати процедуру вирівнювання вікон. Для цього створюються обробники:

```
private void mnuArrangeIcons_Click(object sender,
System.EventArgs e) { this.LayoutMdi(MdiLayout.ArrangeIcons); }

private void mnuCascade_Click(object sender,
System.EventArgs e) { this.LayoutMdi(MdiLayout.Cascade); }

private void mnuTileHorizontal_Click(object sender,
System.EventArgs e) { this.LayoutMdi(MdiLayout.TileHorizontal); }

private void mnuTileVertical_Click(object sender,
System.EventArgs e) { this.LayoutMdi(MdiLayout.TileVertical); }
```

Метод **LayoutMdi** містить перелічення **MdiLayout**, що у свою чергу містить чотири члени. **ArrangeIcons** перемикає фокус на обрану форму, у властивості **MdiList** пункту меню **ArrangeIcons** встановлюється також значення **true**. При відкритті кількох нових документів вікна розташовуються каскадом (рис. 1.6), але вони можуть бути розташовані горизонтально – значення **TileHorizontal**, або вертикально – значення **TileVertical**, але потім знову повернути каскадне розташування – **Cascade** (рис. 1.7).

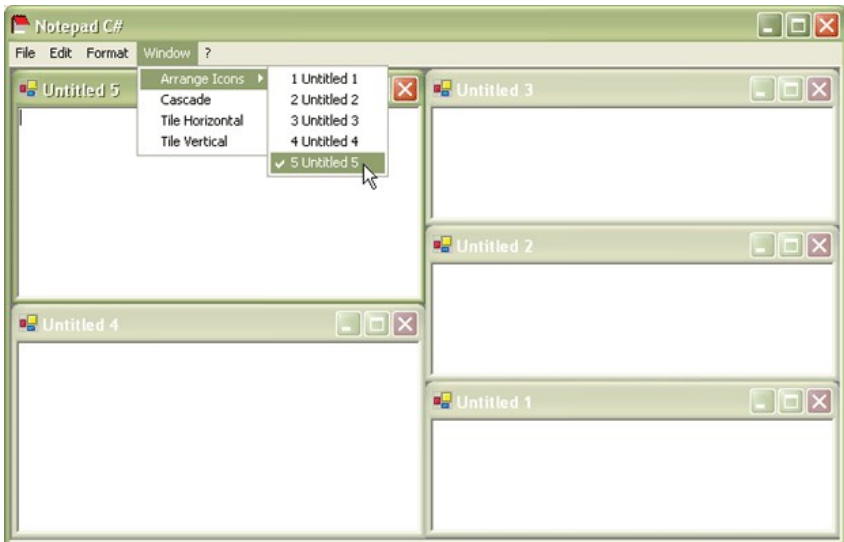


Рисунок 1.7 – Розташування вікон **TileHorizontal** та пункт меню **ArrangeIcons**

1.1.4 Вирізання, копіювання та вставка текстових фрагментів

Із застосунком працювати буде зручніше, якщо при створенні нового документу він одразу буде займати всю область головної форми. Для цього встановлюється властивість **WindowState** форми **blank Maximized**. Тепер створюються обробники для стандартних операцій вирізання, копіювання та вставки. Елемент керування **RichTextBox** має властивість **SelectedText**, яке містить виділений фрагмент тексту. На підставі цієї властивості і будуть

реалізовані дії щодо роботи з текстом. У коді форми **blank** оголошується змінна **BufferText**, у якій буде зберігатися буферизований фрагмент тексту:

```
private string BufferText = "";
```

Надалі створюються відповідні методи:

```
public void Cut() {
    this.BufferText = richTextBox1.SelectedText;
    richTextBox1.SelectedText = "";
}
public void Copy() {
    this.BufferText = richTextBox1.SelectedText;
}
public void Paste() {
    richTextBox1.SelectedText = this.BufferText;
}
public void SelectAll() {
    richTextBox1.SelectAll();
}
public void Delete() {
    richTextBox1.SelectedText = "";
    this.BufferText = "";
}
```

Потім потрібно перемкнутися до режиму дизайну форми та створити обробники для пунктів меню:

```
private void mnuCut_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.Cut();
}
private void mnuCopy_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.Copy();
}
private void mnuPaste_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.Paste();
}
private void mnuDelete_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.Delete();
}
private void mnuSelectAll_Click(object sender, System.EventArgs
e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.SelectAll();
}
```

Властивість **ActiveMdiChild** перемикає фокус на поточну форму, якщо їх відкрито декілька, та викликає один з методів, визначених у дочірній формі.

1.1.5 Контекстне меню

Контекстне меню, яке дублює деякі дії основного меню, – не найшвидший спосіб роботи з програмою, але самий звичний для користувача. Елемент керування **TextBox** містить у собі просте контекстне меню, яке дублює дії підменю **Edit**. Для того, щоб переконатися в цьому, достатньо нанести цей елемент керування на форму та запустити застосунок на виконання (рис. 1.8).

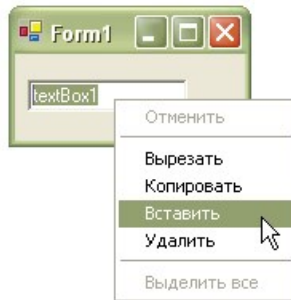


Рисунок 1.8 – Контекстне меню елементу **TextBox**

У застосунку Notepad C# у якості текстового елементу використовується **RichTextBox**. Додається елемент керування **contextMenu** з вікна **ToolBox** на форму **blank**. Додаються пункти контекстного меню точно так само, як це робилося для головного меню (рис. 1.9).

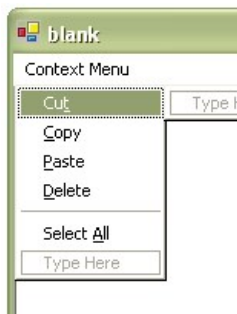


Рисунок 1.9 – Пункти контекстного меню

Властивості **Text** та **Shortcut** пунктів меню потрібно залишити незмінними. Якщо потім буде встановлено для властивості **ShowShortcut** значення **false**, то сполучення клавіш будуть працювати, але в самому меню відображатися не будуть (рис. 1.8). Властивість **Name** буде формуватися наступним чином: для пункту **Cut** – **cmnuCut**, для **Copy** – **cmnuCopy** та інші за аналогією.

У обробнику пунктів викликаються відповідні методи:

```
private void cmnuCut_Click(object sender, System.EventArgs e) {
    Cut();
}
private void cmnuCopy_Click(object sender, System.EventArgs e) {
    Copy();
}
private void cmnuPaste_Click(object sender, System.EventArgs e)
{
    Paste();
}
private void cmnuDelete_Click(object sender, System.EventArgs e)
{
    Delete();
}
private void cmnuSelectAll_Click(object sender,
System.EventArgs e) {
    SelectAll();
}
```

На останнє потрібно визначити, де буде з'являтися контекстне меню. Елемент **RichTextBox** так само, як і форми **frmmain** та **blank**, має властивість **ContextMenu**, де буде змінено **contextMenu1**, оскільки потрібно відображати меню саме у текстовому полі. У запущеному на виконання застосунку – у будь-якому місці тексту буде доступно меню (рис. 1.10).

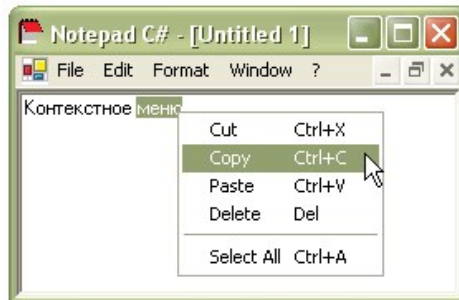


Рисунок 1.10 – Контекстне меню

1.1.6 Діалогові вікна

Середовище Visual Studio. NET містить готові діалогові вікна, які дозволяють обирати чи файл для відкривання чи шлях на диску для збережень поточного файлу (рис. 1.11).

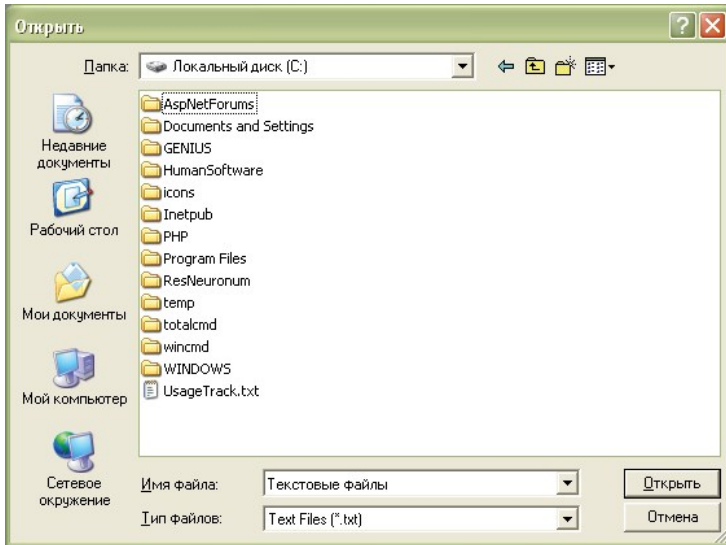


Рисунок 1.11 – Діалогове вікно відкриття файлу

Елемент керування **OpenFileDialog** додається на форму **frmmain** з вікна панелі інструментів **ToolBox**. Подібно елементу **MainMenu**, його буде розташовано на панелі невидимих компонентів (рис. 1.12).

Властивість **FileName** задає назву файлу, яка буде знаходитись у полі "Ім'я файлу:" в момент появи діалогу. Властивість **Filter** задає обмеження до файлів, які можуть бути обрані для відкриття – у вікні буде відображено тільки файли з заданим розширенням. Через вертикальну розділову лінію можна задати зміну типу розширення, яке буде відображено у списку "Тип файлів". Властивість **InitialDirectory** дозволяє задати директорію, звідки буде починатися огляд. Якщо цю властивість не встановлено, початковою директорією буде робочий стіл.

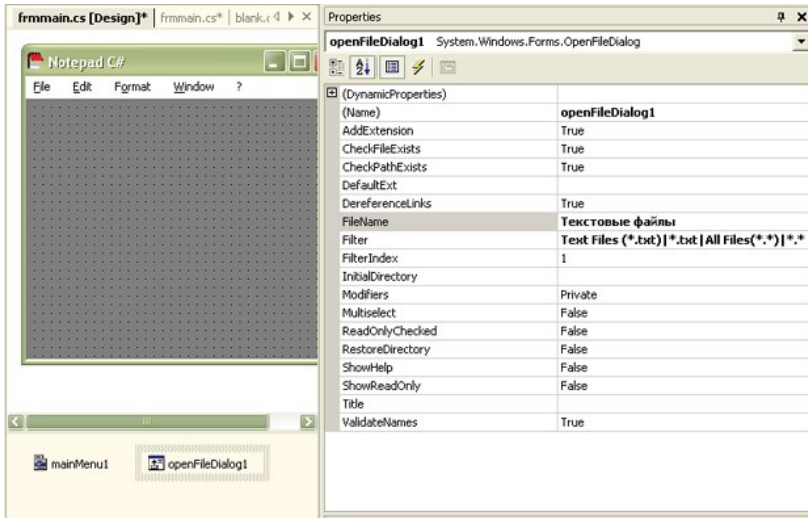


Рисунок 1.12 – Додавання **OpenFileDialog** до форми

Для роботи з файловими потоками в коді форми **blank** ініціюється простір імен **System.IO**:

```
using System.IO;
```

У методі **Open** читається вміст файлу до **RichTextBox**:

```
public void Open(string OpenFileName)
{
    if (OpenFileName == "") {
        return;
    }
    else {
        StreamReader sr = new StreamReader(OpenFileName);
        richTextBox1.Text = sr.ReadToEnd();
        sr.Close();
        DocName = OpenFileName;
    }
}
```

Приклад А.

Додається обробник пункту меню **Open** для форми **frmmain**:

```
private void mnuOpen_Click(object sender, System.EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK) {
        blank frm = new blank();
        frm.Open(openFileDialog1.FileName);
        frm.MdiParent = this;
        frm.DocName = openFileDialog1.FileName;
        frm.Text = frm.DocName;
        frm.Show();
    }
}
```

Приклад Б.

Запускається на виконання програма та відкривається текстовий файл, якій збережено у текстовому форматі (рис. 1.13).

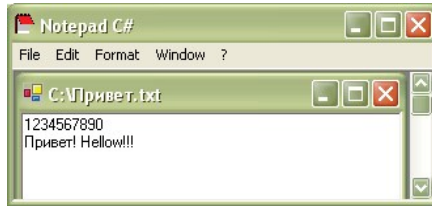


Рисунок 1.13 – Заголовок форми відображає місце розташування та ім'я відкритого файлу

Для коректного відображення кирилиці, текст у редакторі повинно бути збережено у кодуванні Unicode. На жаль, вбудовані діалогові вікна **OpenFileDialog** у середовищі Visual Studio. NET не містять додаткового поля, що дозволяє обирати кодування файлу у момент його відкриття або збереження, як це реалізовано, наприклад, у стандартному застосунку Notepad.

Для збереження файлів використовується елемент керування **SaveFileDialog**, який додається до форми **frmmain**. Властивості цього елементу аналогічні **OpenFileDialog** (рис. 1.12). У початковому коді форми **blank** потрібно внести відповідні зміни:

```
public void Save(string SaveFileName)
{
    if (SaveFileName == "")
        return;
    else {
        StreamWriter sw = new StreamWriter(SaveFileName);
        sw.WriteLine(richTextBox1.Text);
        sw.Close();
        DocName = SaveFileName;
    }
}
```

Приклад В.

Додається обробник пункту меню **Save** для форми **frmmain**:

```
private void mnuSave_Click(object sender, System.EventArgs e)
{
    if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
        blank frm = (blank)this.ActiveMdiChild;
        frm.Save(saveFileDialog1.FileName);
        frm.MdiParent = this;
        frm.DocName = saveFileDialog1.FileName;
        frm.Text = frm.DocName;
    }
}
```

При збереженні внесених змін у вже збереженому файлі замість його перезапису знову з'явиться вікно **SaveFileDialog**. Потрібно змінити програму так, щоб можна було зберігати та перезаписувати файл. Для цього у конструкторі форми **frmmain** після **InitializeComponent** вимикається доступність пункту меню **Save**:

```
mnuSave.Enabled = false;
```

Потім потрібно перейти до режиму дизайну форми **frmmain** та додати пункт меню **Save As** після пункту **Save**. Надалі встановити наступні властивості цього пункту: **Name** – **mnuSaveAs**, **Shortcut** – **CtrlShiftS**, **Text** **Save & As**. У обробнику **Save As** встановлюється обробник пункту **Save** та додається увімкнення доступності **Save**:

```
mnuSave.Enabled = true;
```

Зберігати зміни потрібно як у щойно збережених документах, так і в документах, створених раніше та відкритих для редагування. Тому додається до методу **Open** увімкнення доступності пункту меню **Save**:

```
private void mnuOpen_Click(object sender, System.EventArgs e) {
    mnuSave.Enabled = true;
}
```

У обробнику пункту **Save** додається простий перезапис файлу – виклик методу **Save** форми **blank**:

```
private void mnuSave_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.Save(frm.DocName);
}
```

Тепер, якщо працювати із не збереженими документами, пункт **Save** неактивний (рис. 1.14), але після збереження він стає активним (рис. 1.15) і, крім того, працює поєднання клавіш **Ctrl+S**. Можливо зберігати копію поточного документу, знову скориставшись пунктом меню **Save As** (рис. 1.16).

Описане завдання можна вирішити інакше – поєднавши обидва пункти збереження документу до одного.

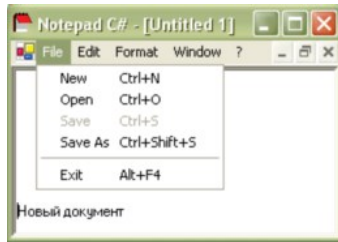


Рисунок 1.14 – Новый документ

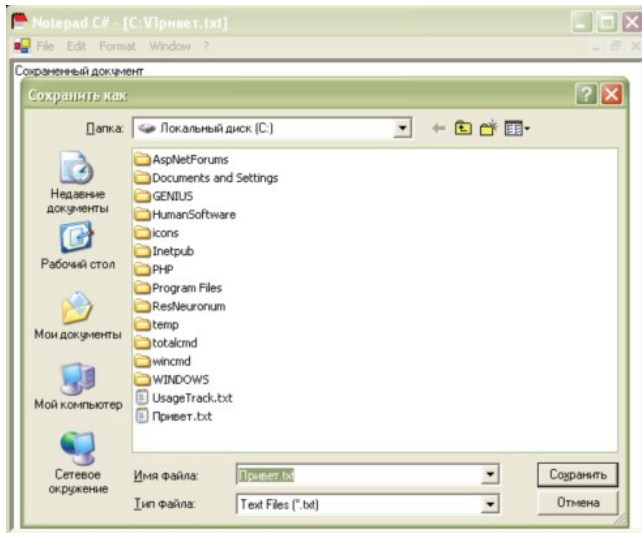


Рисунок 1.15 – Збереження документу

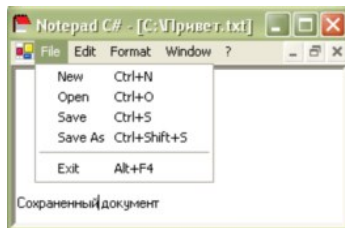


Рисунок 1.16 – Збереження копії

1.1.7 Збереження файлу при закритті форми

Кожного разу, коли потрібно закрити документ, до якого внесено зміни, з'являється вікно попередження, що пропонує зберегти документ. Для реалізації цього потрібно внести аналогічну функцію до застосунку.

У класі **blank**, компоненту **System.Windows.Forms.Form** форми **blank** потрібно створити змінну, яка буде фіксувати збереження документу:

```
public bool IsSaved = false;
```

До обробнику методів **Save** та **Save As** форми **frmmain** додається змінення значення цієї змінної:

```
private void mnuSave_Click(object sender, System.EventArgs e) {
    ...
    frm.IsSaved = true;
}
private void mnuSaveAs_Click(object sender, System.EventArgs e) {
    ...
    frm.IsSaved = true;
}
```

Надалі потрібно перейти до режиму дизайну форми **blank** та у вікні властивостей перемкнутися на події форми (значок з блискавкою). У полі події **Closing** двійним натисканням перейти до коду:

```
private void blank_Closing(object sender,
System.ComponentModel.CancelEventArgs e) {
    if (IsSaved == true)
        if (MessageBox.Show("Do you want save changes in " +
this.DocName + "?", "Message", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
            this.Save(this.DocName);
}
```

Приклад Г.

У момент закриття форми **blank** або **frmmain** з'являється вікно попередження (рис. 1.17):

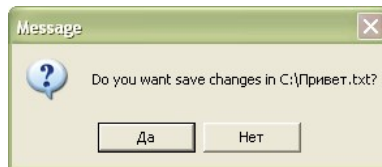


Рисунок 1.17 – Вікно попередження при закритті форми

1.1.8 OpenFileDialog та SaveFileDialog для SDI-застосунків

При створенні MDI-застосунків доводиться розділяти код для відкриття та збереження файлів. У разі SDI-застосунків увесь код буде знаходитись в одному обробнику.

Потрібно створити новий застосунок з назвою **TextEditor**. На основній формі буде розташовано елемент керування **TextBox** та встановлено наступні властивості. Приклади керування **TextBox** наведено у таблиці 1.3.

Таблиця 1.3 – Керування **TextBox**

Властивість	Значення
Name	txtBox
Dock	Fill
Multiline	true
Text	Да

Надалі потрібно додати до форми елемент **MainMenu**, в якому буде всього три пункти – **File**, **Open** та **Save** (властивості цих пунктів надано у таблиці 1.2).

Потім з **ToolBox** необхідно додати елементи **OpenFileDialog** та **SaveFileDialog** – властивості цих елементів такі ж, як у попередній програмі. Надалі у коді форми необхідно додати простір імен для роботи із файловими потоками:

```
using System.IO;
```

Надалі потрібно додати обробник для пункту меню **Open**:

```
private void mnuOpen_Click(object sender, System.EventArgs e) {
    openFileDialog1.ShowDialog();
    string fileName = openFileDialog1.FileName;
    FileStream filestream = File.Open(fileName, FileMode.Open,
FileAccess.Read);
    if (filestream != null) {
        StreamReader streamreader = new StreamReader(filestream);
        txtBox.Text = streamreader.ReadToEnd();
        filestream.Close();
    }
}
```

Приклад Д.

Потім потрібно додати обробник для пункту меню **Save**:

```
private void mnuSave_Click(object sender, System.EventArgs e) {
    saveFileDialog1.ShowDialog();
    string fileName = saveFileDialog1.FileName;
    FileStream filestream = File.Open(fileName, FileMode.Create,
FileAccess.Write);
    if (filestream != null) {
        StreamWriter streamwriter = new StreamWriter(filestream);
        streamwriter.Write(textBox.Text);
        streamwriter.Flush();
        filestream.Close();
    }
}
```

Елемент керування **TextBox** має вбудоване контекстне меню, підтримує клавіші для редагування, а діалогові вікна відкриття та збереження дозволяють працювати із зовнішніми файлами.

1.1.9 FontDialog

Надалі продовжується робота над застосунком Notepad C#. Тепер додається можливість обирати шрифт, його розмір та накреслення. У режимі дизайну поміщується до форми **frmmain** з **ToolBox** елемент керування **FontDialog**. Не змінюючи нічого у властивостях цього елемента, необхідно перейти до обробника пункту **Font** головного меню:

```
private void mnuFont_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.MdiParent = this;
    fontDialog1.ShowColor = true;
    fontDialog1.Font = frm.richTextBox1.SelectionFont;
    fontDialog1.Color = frm.richTextBox1.SelectionColor;
    if (fontDialog1.ShowDialog() == DialogResult.OK) {
        frm.richTextBox1.SelectionFont = fontDialog1.Font;
        frm.richTextBox1.SelectionColor = fontDialog1.Color;
    }
    frm.Show();
}
```

Приклад Ж.

У момент запуску програми на виконання у вікні **Output** з'явиться список помилок (рис. 1.18).

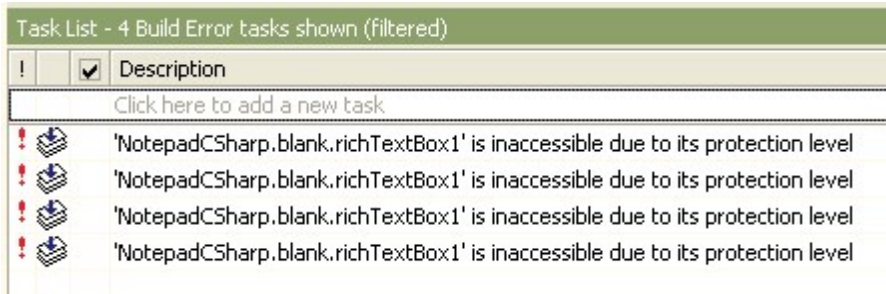


Рисунок 1.18 – Список помилок (NotepadCSharp.blank.richTextBox1 недоступно через свій рівень захисту)

В момент розташування на формі **blank** елементу керування **RichTextBox**, середовище **Visual Studio .NET** згенерувало примірник **richTextBox1** класу **System.Windows.Forms.RichTextBox** із модифікатором доступу **private**, через що, при зверненні до нього і виникає помилка:

```
private System.Windows.Forms.RichTextBox richTextBox1;
```

Тому потрібно змінити модифікатор на **public** та знову запустити на виконання програму. При виборі пункту меню **Font** тепер можна змінювати параметри поточного тексту.

1.1.10 ColorDialog

Діалогове вікно **FontDialog** містить список кольорів, які можуть бути застосовані до тексту, але запропонований список обмежений. Більш цікавою видається можливість призначати користувацький колір, який може бути визначений у великому діапазоні (рис. 1.19):

З панелі **ToolBox** додається елемент керування **ColorDialog** та не змінюючи його властивостей корегується код обробника пункту **Color** головного меню форми **frmmain**:

```
private void mnuColor_Click(object sender, System.EventArgs e) {
    blank frm = (blank)this.ActiveMdiChild;
    frm.MdiParent = this;
    colorDialog1.Color = frm.richTextBox1.SelectionColor;
    if (colorDialog1.ShowDialog() == DialogResult.OK)
        frm.richTextBox1.SelectionColor = colorDialog1.Color;
    frm.Show();
}
```


Код для **ColorDialog** такий же, як і частина коду для властивостей **Color** діалогу **FontDialog**. Це пов'язано з властивостями одного й того ж об'єкту – **RichTextBox**.

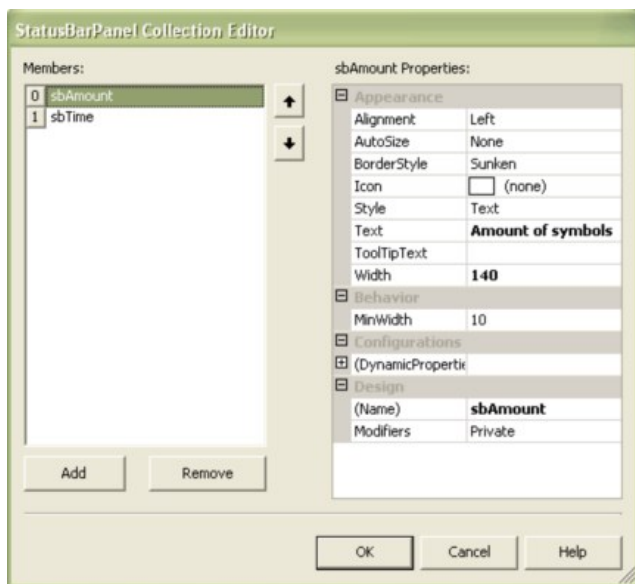
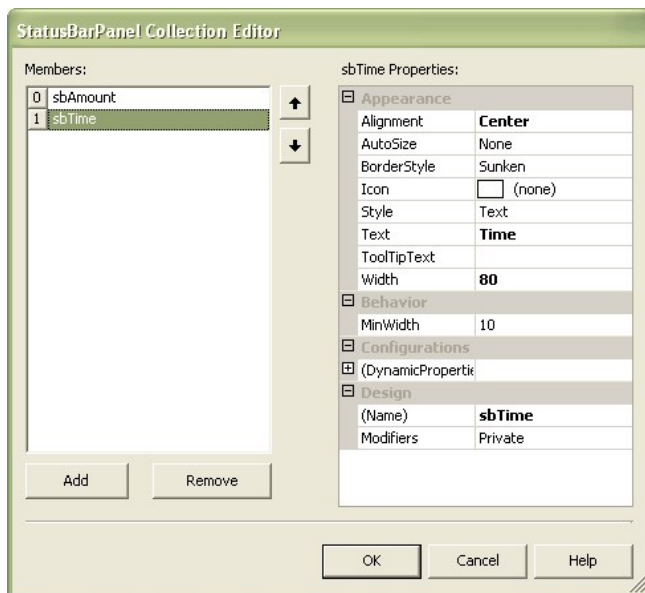


Рисунок 1.19 – Діалогове вікно **ColorDialog**

1.1.11 StatusBar

Елемент керування **StatusBar** застосовується в програмах для відображення інформації в рядок стану – невелику смужку, розташовану внизу програми. У Microsoft Word, наприклад, у статусному рядку відображається кількість сторінок, мова введення, стан перевірки правопису та інші параметри.

Потрібно додати до застосунку Notepad C# рядок стану, на якому буде здійснюватись підрахунок символів, що вводяться та буде відображено системний час. Для цього додається до форми **blank** елемент керування **StatusBar**. Видаляється вміст поля властивості **Text**. В полі властивості **Panels** натискається кнопка «...». У компоненті **StatusBarCollectionEditor** для відображення створюються дві панелі за допомогою натискання на **Add** та для них встановлюються відповідні властивості (змінені значення виділено жирним шрифтом на рис. 1.20 та рис. 1.21).

Рисунок 1.20 – Властивості панелі **sbAmount**Рисунок 1.21 – Властивості панелі **sbTime**

Значення деяких властивостей панелей наводяться у табл. 1.4.

Таблиця 1.4 – Властивості панелей

Властивість	Значення
Alignment	Вирівнювання вмісту Text на панелі
AutoSize	Зміна розмірів панелі за вмістом
BorderStyle	Зовнішній вигляд панелі – утоплена, піднесена або без виділення
Icon	Додавання іконки
Style	Стиль панелі
Text	Текст, що розташовується на панелі
ToolTipText	Спливаюча підказка – з'являється при наведенні курсору на панель
Width	Ширина панелі в пікселях
Name	Назва панелі для звернення до неї в коді

Властивості панелі, які встановлюються у вікні редактору `StatusBarCollectionEditor` можна змінювати в коді.

Після завершення роботи над панелями необхідно вийти з цього редактору. Властивість **ShowPanels** елементу керування **StatusBar** встановлюється у значення **true**. На формі моментально буде відображено дві панелі. Якщо виділити елемент керування **RichTextBox**, то потім у вікні його властивостей необхідно перемкнутися на події та створити обробник для **TextChanged**:

```
private void richTextBox1_TextChanged(object sender,
System.EventArgs e) {
    sbAmount.Text = "Amount of symbols " +
    richTextBox1.Text.Length.ToString();
}
```

Властивість **Text** на панелі **sbAmount** змінюється програмно. У будь-якому випадку у вікні редактору `StatusBarCollectionEditor` при виникненні події **TextChanged** на панелі з'явиться напис.

Для іншої панелі, на котрій буде відображено системний час потрібно у конструкторі форми **blank** додати код:

```

public blank() {
    InitializeComponent();
    sbTime.Text =
Convert.ToString(System.DateTime.Now.ToLongTimeString());
    sbTime.ToolTipText =
Convert.ToString(System.DateTime.Today.ToLongDateString());
}

```

Панель із виведенням часу розташовано досить незвично (рис. 1.22), за необхідністю для виведення часу в звичному правому нижньому куті можна додати третю порожню панель.

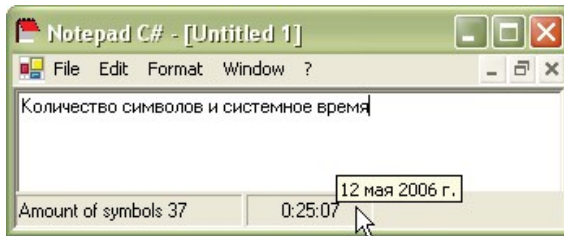


Рисунок 1.22 – Рядок стану з двома панелями

1.1.12 Закривання форми

В момент запуску на виконання програми Notepad C# у заголовку форми розташовано три стандартні кнопки – "Згорнути", "Розгорнути" та "Закрити". Більшість користувачів вважають за краще використовувати саме кнопку "Закрити" для виходу з програми. Проте прийнято дублювати кнопку форми пунктом меню **Exit**. До обробника цієї кнопки необхідно додати код:

```

private void mnuExit_Click(object sender, System.EventArgs e) {
    this.Close();
}

```

Метод **Close** закриває форму та може бути призначений для інших елементів керування.

1.1.13 CheckBox

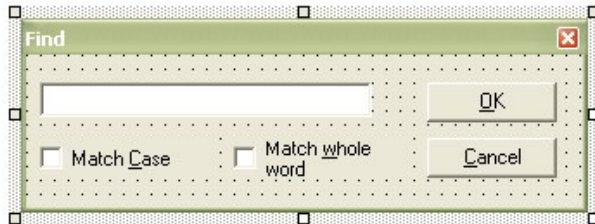
Чекбокси відносяться до так званих кнопок відкладеної дії, тобто їх натискання не запускає негайно будь-який процес. З їх допомогою користувач встановлює певні параметри, результат дії яких позначиться після запуску інших елементів керування.

Для додавання до застосунку Notepad C# форми пошуку заданого тексту, в якій будуть використані елементи керування **CheckBox**, необхідно у вікні Solution Explorer правою кнопкою миші на назві проекту NotepadCSharp обрати пункт Add Windows Form. Потім надати новій формі назву **FindForm.cs** та встановити потрібні властивості (табл. 1.5).

Таблиця 1.5 – Нова форма

FindForm, властивість	Значення
Name	FindForm
FormBorderStyle	FixedToolWindow
Size	328; 112
Text	Find

Тепер форма **FindForm** має дві кнопки та два елементи **CheckBox** (рис. 1.23).

Рисунок 1.23 – Форма **FindForm** у режимі дизайну

Надалі потрібно встановити наступні властивості елементів керування (табл. 1.6 – 1.10)

Таблиця 1.6 – Форма **FindForm, TextBox**

TextBox, властивість	Значення
Name	txtFind
Size	192; 20
Text	

Таблиця 1.7 – Форма **FindForm**, **checkBox1**

checkBox1, властивість	Значення
Name	cbMatchCase
Text	Match &Case

Таблиця 1.8 – Форма **FindForm**, **checkBox2**

checkBox2, властивість	Значення
Name	cbMatchWhole
Text	Match &whole word

Таблиця 1.9 – Форма **FindForm**, **button1**

button1, властивість	Значення
Name	btnOK
DialogResult	OK
Text	&OK

Таблиця 1.10 – Форма **FindForm**, **button2**

button2, властивість	Значення
Name	btnCancel
DialogResult	Cancel
Text	&Cancel

Вибір першого чекбоксу **cbMatchCase** буде встановлювати пошук слів з урахуванням регістру, другого **cbMatchWhole** – пошук за цілим словом. Якщо обрати відповідні властивості кнопок **DialogResult**, то можна встановити обробники для кнопок без звернення до коду. Значення «OK» закриває форму, виконуючи встановлену дію, – пошук, але значення **Cancel** просто закриває форму. У головному меню форми **frmmain** додається розділова лінія та пункт **Find** і встановлюється значення властивостей: **Name** – **mnuFind**, **Shortcut** – **CtrlF**, **Text** – **& Find**. Надалі в обробнику цього пункту:

```
private void mnuFind_Click(object sender, System.EventArgs e) {
    FindForm frm = new FindForm();
    if (frm.ShowDialog(this) == DialogResult.Cancel) return;
```

```

blank form = (blank)this.ActiveMdiChild;
form.MdiParent = this;
int start = form.richTextBox1.SelectionStart;
form.richTextBox1.Find(frm.FindText, start,
frm.FindCondition);
}

```

Основою пошуку буде метод **Find** елементу керування **RichTextBox**. Потрібно лише вказати параметри пошуку, які можуть приймати різні значення (рис. 1.24).

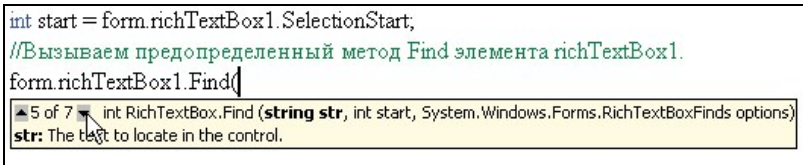


Рисунок 1.24 – Параметри методу **Find**. Якщо клацнути на кнопку підказки, то можна обрати інші підказки для груп переданих параметрів

У коді форми **FindForm** залишилося реалізувати логіку роботи, що залежить від положень елементів **CheckBox**:

```

public RichTextBoxFinds FindCondition {
    get {
        if (cbMatchCase.Checked && cbMatchWhole.Checked)
            return RichTextBoxFinds.MatchCase;
        return RichTextBoxFinds.WholeWord;
        if (cbMatchCase.Checked)
            return RichTextBoxFinds.MatchCase;
        if (cbMatchWhole.Checked)
            return RichTextBoxFinds.WholeWord;
        return RichTextBoxFinds.None;
    }
}

```

Нарешті створюється властивість **FindText**, яка повертає в якості змінної пошуку введений текст до текстового поля форми **FindForm**:

```

public string FindText {
    get
        return txtFind.Text;
    set
        txtFind.Text = value;
}

```

Нарешті можна протестувати програму. Обираючи різні варіанти, можна одноразово шукати задане слово (рис. 1.25).

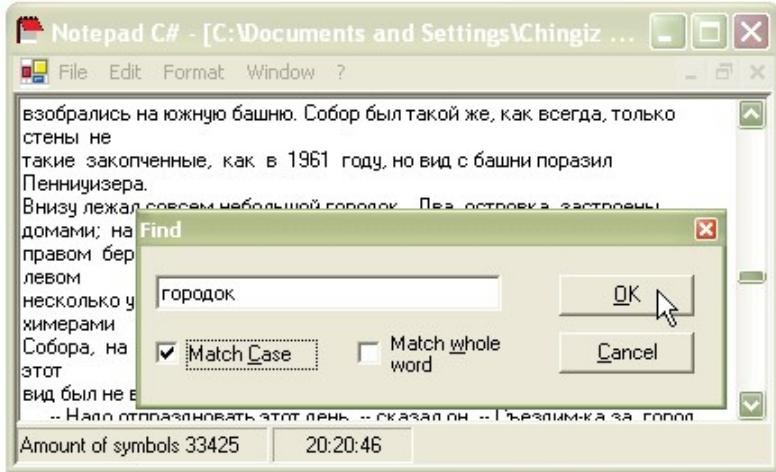


Рисунок 1.25 – Пошук слова в тексті

1.1.14 Властивість TabIndex елементів управління

Багато користувачів вважають за краще працювати по більшій частині з клавіатурою і вдаватися до допомоги миші тільки там, де без неї не можна обійтися. При роботі з формою **Find** було б зручно ввести слово в текстове поле, потім, натиснувши клавішу **Tab** і **Enter**, здійснити пошук, або двічі натиснувши клавішу **Tab**, перемкнути фокус введення на чекбокс **MatchCase**. Для цього у режимі дизайну форми **Find** слід встановити властивість **TabIndex** елементів керування у вигляді послідовної нумерації, починаючи з нуля. Приклади властивості **TabIndex** наведено у табл. 1.7.

Таблиця 1.7 – Властивість **TabIndex**

Елемент керування	TabIndex
TxtFind	0
BtnOK	1
CbMatchCase	2
CbMatchWhole	3
BtnCancel	4

При такій нумерації фокус введення буде послідовно переходити за порядком зростання – від елементу **txtFind** до **btnCancel**. Властивість **TabIndex** має особливо велике значення при створенні ряду текстових полів, де користувачеві буде пропонуватися ввести інформацію.

Програми, як правило, містять пункт головного меню "Про програму", де в окремому вікні відображено логотип компанії, ліцензійна угода, гіперпосилання на сайт розробника та інша інформація. Нова форма створюється з використанням нових елементів керування – **Label**, **LinkLabel** та **PictureBox**.

Отже, до проекту додається нова форма, яка отримує назву **About.cs**. Встановлюються наступні властивості форми (табл. 1.8).

Таблиця 1.8 – Нова форма **About.cs**

About, властивість	Значення
Name	About
FormBorderStyle	FixedSingle
MaximizeBox	False
MinimizeBox	False
Size	318; 214
Text	About Notepad C#

До форми додається елемент керування **PictureBox** – він представляє собою підкладку, яка розміщується на формі та може містити в собі малюнки для відображення. У полі властивості **Image** натискається кнопка «...» та обирається «Рисунок за адресою» ...\\logo.gif. Оскільки **logo.gif** є анімованим малюнком, то елемент **PictureBox** починає відтворювати анімацію відразу ж, навіть у режимі дизайну.

З панелі **ToolBox** перетягується на форму кнопка, **Label** та **LinkLabel**. У полі властивості **Text** кнопки вводиться & OK. Елемент **Label** призначений для розміщення на формі написів, які у готовому застосунку будуть доступні тільки для читання. У полі властивості **Text** вводиться «Notepad C# 2021 All rights reserved».

Елемент **LinkLabel** відображає текст на формі в стилі web-посилань та зазвичай використовується для створення навігації між формами або посиланнями на сайт. У полі **Text** цього елемента вводиться адреса гіпотетичного сайту – www.notepadcsharp.com. Користувач буде здійснювати перехід до сайту, натиснувши на це посилання, тому реалізується перехід за гіперпосиланням для події **Click**. У вікні **Properties** потрібно зробити подвійне натискання на ліву кнопку миші щодо події **Click** та додати обробник:

```
private void linkLabel1_Click(object sender, System.EventArgs e)
{
    try
        VisitLink();
    catch (Exception ex)
        MessageBox.Show(ex + " Unable to open link that was
clicked.");
}
private void VisitLink() {
    linkLabel1.LinkVisited = true;
    System.Diagnostics.Process.Start
("http://www.notepadcsharp.com");
}
```

Кнопка **ОК** просто буде закривати форму:

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

У пункті головного меню **About Programm ...** форми **frmmain** подається процедура виклику форми **About**:

```
private void mnuAbout_Click(object sender, System.EventArgs e) {
    About frm = new About();
    frm.Show();
}
```

Результат роботи форми (рис. 1.26).

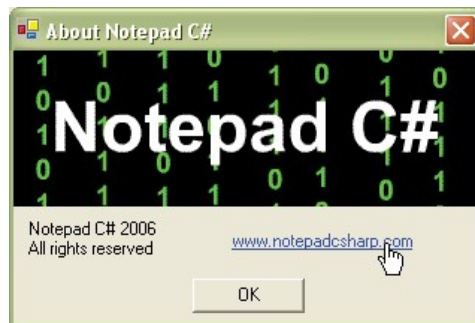


Рисунок 1.26 – Вікно **About Notepad C#**

1.2 Завдання до роботи

Ознайомитися з основними теоретичними відомостями, а також з підходами та методами у розробці текстового редактору на мові C#, використовуючи ці методичні вказівки, а також рекомендовану літературу.

1.3 Хід виконання роботи

Опрацювати та практично реалізувати кожен з прикладів, наведених у цьому розділі. Початковий код створювати самостійно та власноруч, застосовуючи в ньому власні оригінальні рішення. Виконані напрацювання рекомендовано зберегти для подальшого використання у фінальній версії самостійної роботи.

1.4 Контрольні запитання

- 1.4.1 Що таке Елементи керування у C#?
- 1.4.2 Що таке SDI?
- 1.4.3 Що таке MDI?
- 1.4.4 Властивість **ActiveMdiChild** у C#.
- 1.4.5 Застосування **StatusBar**.
- 1.4.6 Метод **LayoutMdi**.
- 1.4.7 Що таке Контекстне меню?
- 1.4.8 Що таке Діалогові вікна?
- 1.4.9 Що таке **OpenFileDialog**?
- 1.4.10 Що таке **SaveFileDialog**?
- 1.4.11 Що таке **FontDialog**?
- 1.4.12 Властивість **TabIndex** елементів керування.
- 1.4.13 Які є способи використання бібліотек?
- 1.4.14 Що таке Доступ до даних?
- 1.4.15 Що таке **ClassView**?
- 1.4.16 Що таке Вікно властивостей?
- 1.4.17 Що таке Панель компонентів?
- 1.4.18 Клас **MessageBox**.
- 1.4.19 Що таке Події C#?

2 РОБОТА З ДРУКОМ ТА СТВОРЕННЯ ЕЛЕМЕНТІВ КЕРУВАННЯ У АЛГОРИТМІЧНІЙ МОВІ C#

2.1 Друк

У будь-якій організації, незалежно від роду її діяльності, часто потрібно друкувати дані та документи. Для цього у .NET Framework передбачено компонент **PrintDocument**, що забезпечує виведення на друк і надає ряд класів для підтримки друку та налаштування пов'язаних з цим параметрів. Розробникам програмного забезпечення необхідно створювати підтримку друку в своїх застосунках.

2.1.1 Компонент **PrintDocument**

У компоненті **PrintDocument** немає графічного інтерфейсу, але цей компонент знаходиться на вкладці Windows Forms інструментальної панелі Toolbox, звідки його можна перетягнути до форми застосунку, яку було відкрито у вікні дизайнера. Об'єкт **PrintDocument** інкапсулює всю інформацію, необхідну для друку сторінок. У нього є ряд властивостей:

- **PrinterSettings** зберігає відомості про можливості та конфігурації доступних принтерів;
- **DefaultPageSettings** інкапсулює параметри друку сторінок;
- **PrintController** контролює обробку сторінок при друці.

Створення об'єкту **PrintDocument**

Об'єкт **PrintDocument** можна створити під час розробки за допомогою графічного інтерфейсу дизайнера, або програмно під час виконання.

У першому випадку об'єкт **PrintDocument** слід перетягнути до вікна дизайнера безпосередньо із панелі Toolbox, та серед компонентів з'явиться новий екземпляр цього об'єкту, який буде автоматично конфігуровано для роботи із системним принтером за замовчуванням.

У другому випадку екземпляр об'єкту **PrintDocument** створюється за прикладом:

```
PrintDocument myPrintDocument = new PrintDocument();
```

Отриманий таким чином примірник об'єкту **PrintDocument** автоматично конфігурується для роботи з системним принтером за замовчуванням.

Підтримка друку

У моделі друку, яка прийнята в .NET Framework, дані друкуються безпосередньо але передаються кодом програми. Виклик методу **Print** ініціює нове завдання друку, одночасно генеруючи подію **PrintPage**. Якщо клієнтський обробник для цієї події відсутній, то друк не виконується. Щоб задати дані для друку, необхідно створити обробник для події **PrintPage**.

Якщо завдання друку складається з декількох сторінок, подія **PrintPage** генерується для кожної з них, тому обробник цієї події викликається багатократно. Оброблювач події **PrintPage** контролює виконання друку багатосторінкового документу.

Подія **PrintPage**

Щоб реально відправити дані на принтер, необхідно створити обробник для події **PrintPage** та додати до нього код, що сформує друкований вміст. Усі дані та об'єкти, необхідні для друку, передаються до обробника в об'єкті **PrintPageEventArgs**, властивості якого описано у таблиці 2.1.

Таблиця 2.1 – Властивості об'єкта **PrintPageEventArgs**

Ім'я	Опис
Cancel	Вказує, чи скасовано завдання друку.
Graphics	Містить об'єкт Graphics, що формує друковану сторінку.
HasMorePages	Отримує або встановлює значення, яке вказує, чи всі сторінки надруковано.
MarginBounds	Отримує об'єкт Rectangle, що представляє область сторінки, обмежену полями.
PageBounds	Отримує об'єкт Rectangle, що представляє сторінку цілком.
PageSettings	Отримує або встановлює об'єкт PageSettings для поточної сторінки.

Для перетворення вмісту документу до даних для принтеру застосовується об'єкт **Graphics**, який доступний через клас **PrintPageEventArgs**. У цьому випадку сторінка, яка друкується відіграє ту саму роль, що й клієнтська область форми елементу керування на будь-якій іншій поверхні для відображення, наданої об'єктом **Graphics**. Друковану сторінку формують тими самими методами, які застосовуються для відображення вмісту форми на екрані. Приклад дуже простого методу, який друкує еліпс, вписаний до області, обмежену полями:

```
//Щоб цей метод був викликаний, він повинен оброблятися подією
//PrintPage.
public void PrintEllipse(object sender,
System.Drawing.Printing.PrintPageEventArgs e) {
    e.Graphics.DrawEllipse(Pens.Black, e.MarginBounds);
}
```

Властивості **MarginBounds** та **PageBounds** вказують області сторінки, які доступні для друку. Межами області друку можна зробити поля сторінки, обчисливши координати друкованих об'єктів на основі прямокутника **MarginBounds**. Можливо друкувати і за межами сторінкових полів, наприклад, щоб створити верхній та нижній колонтитули. Для цього слід розрахувати координати об'єктів на основі прямокутника **PageBounds**. Як і при виведенні на дисплей, при друкуванні координати за замовчуванням обчислюються у пікселях.

Властивість **HasMorePages** дозволяє вказати, чи є завдання друку багатосторінковим. За замовчуванням цю властивість встановлено у **false**. Якщо ж буде потрібно надрукувати багатосторінковий документ, то слід встановити це в **true**, а по завершенні друку останньої сторінки – знову в **false**.

Обробник події **PrintPage** повинен стежити за кількістю сторінок заданим для друку, в іншому випадку результати будуть непередбачуваними. Якщо після закінчення друку останньої сторінки не встановити властивість **HasMorePages** у **false**, застосунок буде генерувати події **PrintPage** знову й знову.

Крім того, передбачено можливість скасувати завдання друку, не чекаючи закінчення виведення поточної сторінки. Для цього слід встановити властивість **Cancel** у **true**.

Для створення обробника події **PrintPage** потрібно виконати подвійне натискання на елемент **PrintDocument** у вікні дизайнера (створення обробника за замовчуванням) або виконати створення обробника вручну.

2.1.2 Виведення на друк

Друк графіки

Друкувати графічні елементи не складніше, ніж створювати їх на екрані дисплею. Для обох цілей застосовують об'єкт **Graphics**, який доступний через об'єкт **PrintPageEventArgs**. Друкують як прості фігури, так і більш складні. У останньому випадку використовують об'єкт **GraphicsPath**. Приклад, як надрукувати складну фігуру за допомогою об'єкту **GraphicsPath**:

```
//Цей метод повинна обробляти подія PrintPage
public void PrintGraphics(object sender,
    System.Drawing.Printing.PrintPageEventArgs e) {
    System.Drawing.Drawing2D.GraphicsPath myPath = new
    System.Drawing.Drawing2D.GraphicsPath();
    myPath.AddPolygon(new Point[] {new Point(1,1),
    new Point(12,55), new Point(34,8), new Point(52,53),
    new Point(99,5)});
    myPath.AddRectangle(new Rectangle(33,43,20,20));
    e.Graphics.DrawPath(Pens.Black, myPath);
}
```

Щоб надрукувати багатосторінковий графічний документ, потрібно вручну розбити його на сторінки та створити відповідний код. Наступний метод має при друкуванні розбити еліпс на дві сторінки, який за розмірами не вміщується на одній сторінці:

```
bool FirstPagePrinted = false;
public void PrintBigEllipse(object sender,
    System.Drawing.Printing.PrintPageEventArgs e) {
    if (FirstPagePrinted == false) {
        FirstPagePrinted = true;
        e.HasMorePages = true;
        e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0,0,
        e.PageBounds.Width, e.PageBounds.Height*2));
    }
    else {
        e.HasMorePages = false;
        FirstPagePrinted = false;
        e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0,
        -(e.PageBounds.Height), e.PageBounds.Width, e.PageBounds.Height*2));
    }
}
```

У цьому прикладі виведення сторінок на друк реалізовано за допомогою змінної **FirstPagePrinted**. Ця змінна оголошена поза методу. Якщо оголосити її усередині методу, вона буде ініціалізована при кожному його виклику та завжди буде повертати **false**. Дійсно, під час друку кожної сторінки, програма відновлює еліпс заново, змінюючи його положення так, щоб на поточну сторінку потрапила відповідна частина фігури.

Друк тексту

Виведення на друк тексту відбувається так само, як до екрану дисплея, – за допомогою методу **Graphics.DrawString**. Як і при виведенні до екрану, необхідно вказати шрифт, сам текст, об'єкт **Brush** та координати початку друку, наприклад:

```
Font myfont = new Font("Batang", 36, FontStyle.Regular,
GraphicsUnit.Pixel);
string Hello = "Hello World!";
e.Graphics.DrawString(Hello, myfont, Brushes.Black, 30, 30);
```

Перед друком тексту слід програмно перевірити, чи не виходить якась його частина за межі сторінки, – будь-який вміст, який виходить за межі сторінки, надруковано не буде.

Друк багаторядкового тексту

Для друку багаторядкового тексту, наприклад масиву рядків або вмісту текстів файлу, необхідний код, який обчислює відстань між окремими рядками. Щоб обчислити кількість рядків на сторінці, необхідно розділити висоту клієнтської області сторінки (вона дорівнює висоті сторінки за вирахуванням полів) на кегль шрифту. Розташування будь-якого рядку на сторінці можна обчислити, помноживши її номер на кегль шрифту. Наступний приклад демонструє друк рядкового масиву **myStrings**:

```
//Потрібна змінна, яка відстежує поточний елемент масиву
//Щоб уникнути ініціалізації цієї змінної при друці кожної
//сторінки, слід її оголошувати поза обробнику події PrintPage
int ArrayCounter = 0;
//Це обробник події PrintDocument.PrintPage. Він припускає,
//що рядковий масив myStrings вже оголошено та заповнено,
//а також, що ініціалізований об'єкт myFont - обраний шрифт
private void PrintStrings(object sender, PrintPageEventArgs e) {
    //Змінні, які керують міжрядковими інтервалами документу
    float LeftMargin = e.MarginBounds.Left;
    float TopMargin = e.MarginBounds.Top;
```



```

float MyLines = 0;
float YPosition = 0;
int Counter = 0;
string CurrentLine;
//Обчислення кількості рядків на сторінці
MyLines = e.MarginBounds.Height /
myFont.GetHeight(e.Graphics);
//Друк усіх рядків файлу із зупинкою в кінці сторінки
while (Counter < MyLines &&
ArrayCounter <= myStrings.GetUpperBound()) {
    CurrentLine = myStrings[ArrayCounter];
    YPosition = TopMargin + Counter *
myFont.GetHeight(e.Graphics);
    e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black,
LeftMargin, YPosition, new StringFormat());
    Counter++;
    ArrayCounter++;
    //Якщо надруковано не всі рядки, то друк наступної сторінки
    if (!(ArrayCounter == myStrings.GetUpperBound(0)))
        e.HasMorePages = true;
    else
        e.HasMorePages = false;
}
}

```

Робота з кольорами

Набір параметрів для друку на принтері, що підтримує кольори, відрізняється від того, що потрібно для друку на чорно-білому принтері. При кольоровому друкуванні необхідно врахувати, що деякі кольори, які добре помітні на екрані, практично не помітні на папері. Таким чином, код для друку на чорно-білих та кольорових принтерах не однаковий.

Щоб визначити, чи підтримує поточний принтер кольоровий друк, перевірте властивість **PrinterSettings.SupportsColor**. Якщо це так, властивість **DefaultPageSettings.Color** встановлено в **true**, а принтер буде налаштовано на кольоровий друк за замовчуванням. Для чорно-білого друку встановіть властивість **DefaultPageSettings.Color** у **false**.

Наступний приклад демонструє організацію підтримки кольорового та чорно-білого друку за допомогою цієї властивості:

```

Brush BrushOne, BrushTwo;
if (PrintDocument1.PrinterSettings.SupportsColor == true) {
    //Створити пензлі для кольорового друку
    BrushOne = Brushes.Red;
    BrushTwo = Brushes.Blue;
}

```

```

else {
    //Створити об'єкти HatchBrush для чорно-білого друку
    BrushOne = new HatchBrush(HatchStyle.DashedVertical,
    Color.Black);
    BrushTwo = new HatchBrush(HatchStyle.DashedHorizontal,
    Color.Black);
}

```

Організація підтримки друку в папці:

- необхідно перетягнути об'єкт **PrintDocument** з панелі **Toolbox** до вікна дизайнеру, – новий екземпляр з'явиться у області компонентів, автоматично налаштований для використання принтеру за замовчуванням. Але створити компонент **PrintDocument** можна і програмно;

- необхідно створити обробник для подій **PrintDocument** та **PrintPage**;

- необхідно додати обробнику події **PrintPage** код, який формує вміст у клієнтській області сторінки. Надалі потрібно передати сформоване зображення до принтеру за допомогою об'єкту **PrintPageEventArgs.Graphics**;

- якщо потрібно працювати з багатосторінковими документами, то створюється код, що керує розбиттям документу на сторінки.

2.1.3 Застосування елементу **PrintPreviewControl**

Існує елемент керування **PrintPreviewControl**, що забезпечує можливість попереднього перегляду документу перед відправкою його на друк. Він знаходиться на вкладці Windows Forms інструментальної панелі **Toolbox**, звідки його можна перетягнути на форму. Щоб налаштувати попередній перегляд сторінки, потрібно пов'язати елемент керування **PrintPreviewControl** з примірником **PrintDocument** наступним чином:

```
myPrintPreview.Document = myPrintDocument;
```

Після цього елемент керування **PrintPreviewControl** буде відображати вміст, призначений для друку. Потім потрібно перехопити обробником події **PrintDocument.Print** сформований їм графічний образ сторінки та відобразити його в елементі керування **PrintPreviewControl**.

При зміні параметрів програми часто змінюється і вигляд друкованого документу.

Для оновлення образу сторінки, що відображається при попередньому перегляді, викликається метод **InvalidatePreview**:

```
myPrintPreview.InvalidatePreview();
```

Zoom – ще одна важлива властивість елемента керування **PrintPreviewControl**, вказує масштаб сторінки при попередньому перегляді. Якщо дорівнює **1**, то масштаб дорівнює **100%**. При значеннях до **1**, зображення для попереднього перегляду менше оригіналу, а при значеннях, що перевищують **1**, зображення відповідно більше.

Елемент керування **PrintPreviewDialog**

Елемент керування **PrintPreviewControl** істотно полегшує створення підтримки попереднього перегляду документів, що друкуються, у тому числі додавання нестандартних режимів перегляду. Але більшості застосункам цілком достатньо стандартних функцій попереднього перегляду при друці. Для цих цілей передбачено елемент керування **PrintPreviewDialog**, який підтримує найпоширеніші можливості попереднього перегляду при друці, відображає діалогове вікно, яке наведено на рисунку 2.1.

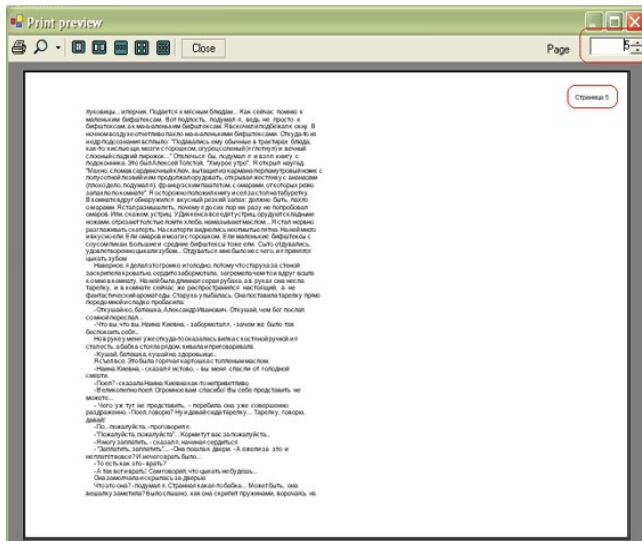


Рисунок 2.1 – Вікно **PrintPreviewDialog**

Для попереднього перегляду документу слід помістити у властивість **Document** об'єкту **PrintPreviewDialog** об'єкт **PrintDocument**, який представляє потрібний документ. Елемент керування **PrintPreviewDialog** дозволяє переглядати кілька сторінок документу одночасно, змінювати масштаб та відправляти документ на друк. Виклик вікна **PrintPreviewDialog** не відрізняється від виклику будь-якого іншого вікна та здійснюється за допомогою методу **Show** або **ShowDialog**.

2.1.4 Налаштування параметрів друку

Інтерфейс друку .NET Framework підтримує безліч керуючих параметрів. Властивості **PrintDocument** та **PrinterSettings** містять відомості про принтери, які доступні у системі. Властивість **PrintDocument.DefaultPageSettings** – параметри сторінки за замовчуванням (вони застосовуються, якщо в обробнику події **PrintPage** не вказано інших параметрів). У свою чергу, у цих властивостей є безліч власних властивостей. Призначення більшості з них зрозуміло без додаткових пояснень. Крім того, при створенні об'єкту **PrintDocument** до властивості **PrinterSettings** поміщується об'єкт принтеру за замовчуванням, який конфігуровано параметрами за замовчуванням, а до властивості **DefaultPageSettings** записуються параметри сторінки за замовчуванням. Таким чином, для створення і виконання завдання друку можна обійтися параметрами за замовчуванням.

Якщо необхідно дозволити користувачам самостійно конфігурувати завдання друку, то потрібно скористатися елементами керування Visual Studio .NET, що забезпечує безкоштовну інтеграцію налаштування принтеру з призначеним для користувача інтерфейсом застосувань **PrintDialog** та **PageSetupDialog**.

Застосування **PrintDialog**

Вікно **PrintDialog** дозволяє встановити властивість **PrinterSettings** об'єкту **PrintDocument** у період виконання. Щоб додати екземпляр **PrintDialog** до застосунку, потрібно перетягнути його із вкладки Windows Forms панелі **Toolbox** до вікна конструктора. Щоб відобразити вікно **PrintDialog** у період

виконання, необхідно викликати метод **ShowDialog** наступним чином:

```
PrintDialog1.ShowDialog();
```

Вікно **PrintDialog** необхідно пов'язати з об'єктом **PrintDocument**. Для цього поміщується до властивості **PrintDialog.Document** об'єкт **PrintDocument**, який представляє потрібний документ.

Під час відображення, вікно **PrintDialog** (рис. 2.2) пов'язується з властивістю **PrinterSettings** об'єкту **PrintDocument**, заданого властивістю **Document**, та надається користувачеві графічний інтерфейс для налаштування друку.

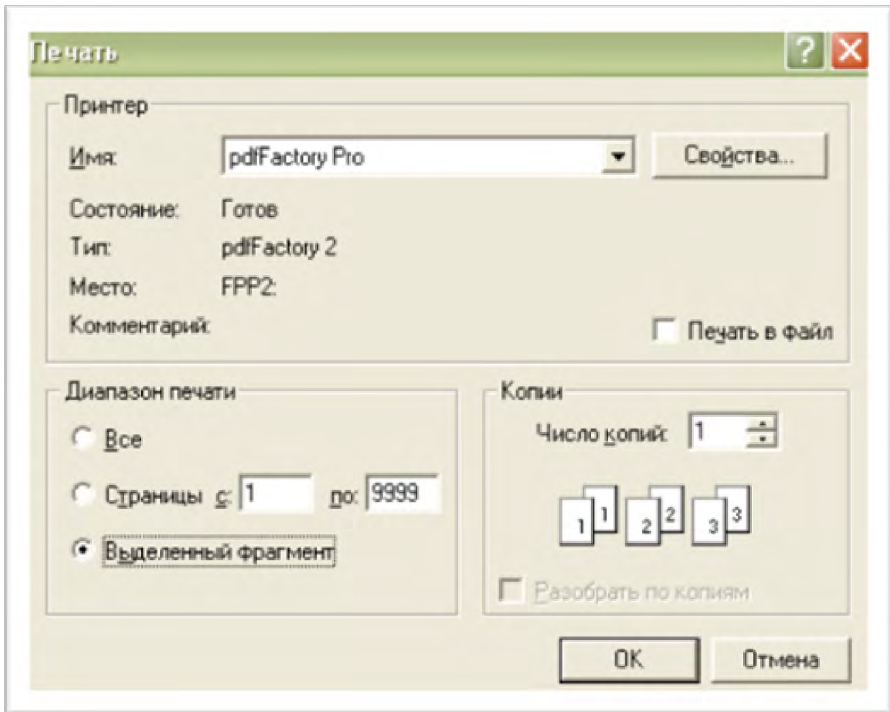


Рисунок 2.2 – Вікно **PrintDialog**

Застосування PageSetupDialog

Методики застосування вікон **PageSetupDialog** та **PrintDialog** подібні. Вікно **PageSetupDialog** (рис. 2.3) надає графічний інтерфейс для налаштування параметрів сторінки та друку у період виконання. Користувач може обрати орієнтацію сторінки, розміри паперу та встановити поля, а також налаштувати принтер.

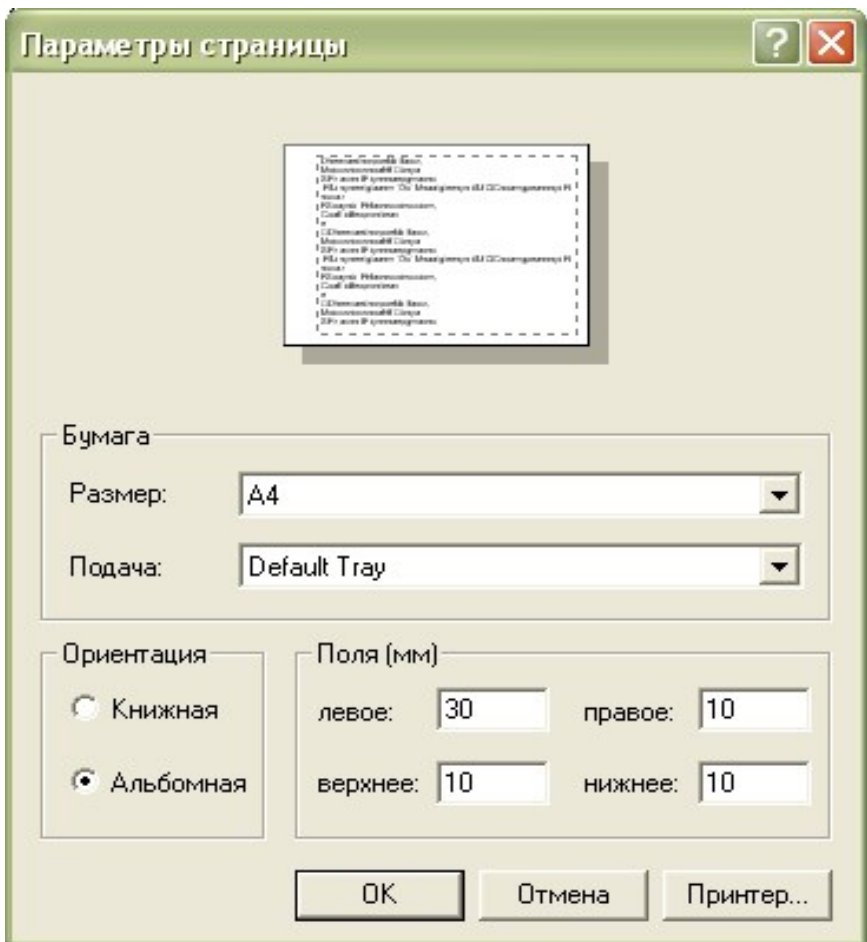


Рисунок 2.3 – Вікно **PageSetupDialog**

Як і **PrintDialog**, вікно **PageSetupDialog** підтримує властивість **Document**, яка містить об'єкт **PrintDocument**, що представляє потрібний документ. Встановивши властивість **Document**, можна зв'язати об'єкти **PrintDocument** та **PageSetupDialog**. При цьому будь-які зміни, зроблені у вікні налаштування друку, будуть відображатися у властивостях об'єктів **PrintDocument**, **PrinterSettings** і **DefaultPageSettings**.

Конфігурування **PageSettings** у період виконання

Іноді окремі сторінки багатосторінкового документу потрібно надрукувати не так, як інші, наприклад, вивести одну сторінку в альбомній орієнтації, а всі інші – у книжковій. Змінити параметри окремої сторінки дозволяє властивість **PrintPageEventArgs**. **PageSettings**, яка надає параметри друку сторінки. Зміни параметрів друку, зроблені за допомогою цієї властивості, діють тільки на поточну сторінку, але інші сторінки завдання друкуються, як задано при відправленні документу на друк. Наступний приклад демонструє зміни макету в період виконання, встановлюючи альбомну орієнтацію сторінки:

```
PrintPageEventArgs.e.PageSettings.Landscape = true;
```

Конфігурування **PageSettings** у період виконання:

- документи, які друкуються представлені екземпляром класу **PrintDocument**. Цей клас підтримує властивості **PrinterSettings**, який задає параметри принтеру, та **DefaultPageSettings**, що визначає параметри сторінки за замовчуванням;

- друк документу виконують шляхом виклику методу **PrintDocument.Print**. Виклик цього методу генерує подія **PrintPage**, до обробнику якого поміщують початковий код для промальовування документу;

- об'єкт **PrintPageEventArgs** містить усі дані та підтримує всі функції, необхідні для промальовування вмісту документу на принтері. Дані передаються на принтер до об'єкту **Graphics**, що представляє цей принтер;

- об'єкт **Graphics** передається через об'єкт **PrintPageEventArgs**;

- для друку багатосторінкових документів до застосунку необхідно додати початковий код, що виконує розбиття документу на сторінки. Для повторної генерації події **PrintPage** потрібно встановити властивість **PrintEventArgs.HasMorePages** у **true**;
- елемент керування **PrintPreviewControl** забезпечує попередній перегляд документів перед відправкою на друк, а **PrintPreviewDialog** надає найбільш затребувані можливості **PrintPreview** у зручній для використання формі;
- вікна **PrintDialog** та **PageSetupDialog** дають користувачам можливість налаштовувати принтер і параметри сторінки в період виконання. Передбачено також можливість налаштування параметрів окремої сторінки – для цього достатньо змінити властивість **PrintPageEventArgs.PageSettings** під час друку.

2.1.5 Приклад друку вмісту різних компонентів

На формі потрібно розташувати компоненти **RichTextBox** та **MenuStrip** (табл. 2.2).

Таблиця 2.2 – Пункти головного меню

Name	Text	Shortcut
mnuFile	&Файл	
Name	Text	Shortcut
mnuOpen	&Відкрити	Ctrl+O
mnuSave	&Зберегти	Ctrl+S
menuItem1		
mnuPageSetup	Пара&метри сторінки	
mnuPrintPreview	Попер&едній перегляд	
mnuPrint	&Друк	Ctrl+P

Надалі потрібно перетягнути до форми з вікна **ToolBox** елементи керування: **PrintDocument**, **PageSetupDialog**, **PrintPreviewDialog** та **PrintDialog**. Подібно іншим елементам діалогів, вони відображаються на панелі компонентів у середовищі Visual Studio. NET. При друкуванні формується одна або кілька сторінок, за які відповідає об'єкт **PrintDocument**. Елементи

керування **PageSetupDialog**, **PrintPreviewDialog** і **PrintDialog** представляють собою діалогові вікна параметрів сторінок, перегляду та друку відповідно. Налаштування властивостей цих об'єктів є настільки гнучким і широким завданням, яке реалізується в початковому коді програми, що на відміну елементів **OpenFileDialog** або **SaveFileDialog**, має сенс створювати їх програмно в класі форми:

```
PrintDocument printDocument1 = new PrintDocument();
PageSetupDialog pageSetupDialog1 = new PageSetupDialog();
PrintPreviewDialog printPreviewDialog1 =
new PrintPreviewDialog();
PrintDialog printDialog1 = new PrintDialog();
```

Цей фрагмент повністю аналогічний додаванню до форми елементів.

У властивості **Document** елементів керування **PageSetupDialog** та **PrintPreviewDialog** встановити значення властивості **Name** елементу **PrintDocument** – **printDocument1**. Так буде пов'язано об'єкт **printDocument1** з діалоговими вікнами, що відповідає за формування сторінок документу. Надалі потрібно встановити наступні властивості елементу **PrintDialog** (табл. 2.3).

Таблиця 2.3 – Властивості елементу **PrintDialog**

printDialog1, властивість	Значення	Опис
AllowSelection	True	Дозвіл на друк виділеного фрагменту документа
AllowSomePages	True	Дозвіл на друк кількох сторінок
PrintDocument	printDocument1	Зв'язування з екземпляром об'єкту PrintDocument

На інформаційній панелі вікна **Properties** при виборі властивостей **AllowSelection** або **AllowSomePages** розташовано повідомлення – **Enables and disables the Selection/Pages radio button** (Вмикання або вимикання доступності перемикачів "Виділений фрагмент/Сторінки"). Саме по собі визначення цих властивостей не містить відповідну функціональність – це ще одна причина, за якою конфігурувати об'єкти друку краще програмно:

```
InitializeComponent();
PageSetupDialog1.Document = printDocument1;
PrintPreviewDialog1.Document = printDocument1;
PrintDialog1.Document = printDocument1;
PrintDialog1.AllowSomePages = true;
PrintDialog1.AllowSelection = true;
```

У коді форми для роботи з друком в бібліотеці .NET Framework застосовується клас **System.Drawing.Printing**, який потрібно підключити на самому початку роботи:

```
using System.Drawing.Printing;
```

У класі форми оголошуються наступні змінні:

```
//Змінна щодо зберігання тексту для друку з RichTextBox:
string stringPrintText;
//Змінна, яка визначає номер сторінки, з якої починається друк:
int StartPage;
//Змінна, яка визначає кількість сторінок для друку:
int NumPages;
//Змінна, яка визначає номер поточної сторінки:
int PageNumber;
```

Слід звернути увагу на змінну **stringPrintText** – саме вона буде відповідати за передання тексту до об'єкту друку. Оскільки для неї визначено рядковий тип, то не має можливості виводити до друку малюнки в тексті, адже елемент **RichTextBox** підтримує їх наявність (у цьому можна переконатися, скопіювавши вміст документу Microsoft Word до створюваного застосунку).

У конструкторі головної форми потрібно визначити діапазон сторінок для друку:

```
public Form1() {
    InitializeComponent();
    ...
    //Номер сторінки, з якої слід почати друк:
    printDialog1.PrinterSettings.FromPage = 1;
    //Кінцевий номер друкованої сторінки:
    printDialog1.PrinterSettings.ToPage =
printDialog1.PrinterSettings.MaximumPage;
}
```

Максимальна кількість сторінок має діапазон 1-9999. Основною подією, в обробнику якої практично і буде здійснюватися весь друк, буде подія **PrintPage** елементу **printDocument1**. Для цього потрібно перейти до режиму дизайну, виділити елемент та у вікні **Properties** перемкнутися на вкладку подій:

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e) {
}
```

Надалі всі фрагменти коду необхідно додати у цей обробник. Клас **Graphics**, що належить простору імен **System.Drawing** – один з найважливіших класів, який відповідає за відображення графіки та виведення тексту на формі. При створенні нової форми простір імен **System.Drawing** за замовчуванням додається до шаблону. Для формування сторінки друку потрібно створити екземпляр цього класу:

```
Graphics graph = e.Graphics;
//Об'єкт font містить шрифт елементу rtbText
Font font = rtbText.Font;
//Значення міжрядкового інтервалу - висота шрифту
float HeightFont = font.GetHeight(graph);
//Екземпляр stringformat класу StringFormat потрібен
//для визначення додаткових параметрів форматування тексту
StringFormat stringformat = new StringFormat();
```

Сторінка представляє собою прямокутний об'єкт, параметри якого слід визначити. Для цього використовується клас **RectangleF** (літера **F** вказує на використання типу даних з плаваючою точкою **float**):

```
//Створюються екземпляри rectanglefFull та rectfText класу
//RectangleF для визначення областей друку і тексту:
RectangleF rectanglefFull, rectfText;
//Створюються змінні для підрахунку числа символів та рядків:
int NumberSymbols, NumberLines;
```

Перша область – повний розмір сторінки, який містить назви, наприклад, формат **A4** або **A3**. Інформацію щодо цієї області надає властивість **PageBounds** класу **PrintPageEventArgs**. Значенням цієї властивості є об'єкт **Rectangle**, властивості **X** та **Y** якого дорівнюють **0**, а властивості **Width** та **Height** надають розміри сторінки паперу за замовчуванням, які виражено в сотих частках дюйма. Друга область пов'язана з конструктивними особливостями принтерів – більшість з них використовують для друку площею, меншою за розміри паперу. Область друку визначається значенням властивості **VisibleClipBounds** класу **Graphics**, яка надається об'єктом **RectangleF**. Властивості **X** та **Y** цього об'єкту мають значення **0**, а її властивості **Width** та **Height** дорівнюють горизонтальному та вертикальному розмірам області друку сторінки. У третій області міститься текст, яка зазвичай визначається користувачем. За замовчуванням, область відступів від краю розраховується з розміром в один дюйм за периметром сторінки. Цей

розмір повертається об'єктом **Rectangle** властивості **MarginBounds**. У будь-якому випадку, області формуються об'єктами **Rectangle** або **RectangleF**, конструктори яких можуть мати наступний вигляд:

```
Rectangle(int x, int y, int width, int height);
RectangleF(float x, float y, float width, float height);
```

Ці конструктори відрізняються тільки типом даних, які формують структуру. Наприклад, для виділення області друку потрібно встановити об'єкт **rectanglefFull**:

```
if (graph.VisibleClipBounds.X < 0)
    rectanglefFull = e.MarginBounds;
else
    rectanglefFull = new RectangleF(e.MarginBounds.Left -
    (e.PageBounds.Width - graph.VisibleClipBounds.Width)/2,
    e.MarginBounds.Top - (e.PageBounds.Height -
    graph.VisibleClipBounds.Height)/2, e.MarginBounds.Width,
    e.MarginBounds.Height);
    //Область тексту буде представляти собою копію області
    //rectanglefFull з урахуванням висоти шрифту:
    rectanglefText = RectangleF.Inflate(rectanglefFull, 0,
    -2 * HeightFont);
    //Кількість рядків:
    int NumDisplayLines = (int)Math.Floor(rectanglefText.Height /
    HeightFont);
    //Висота області:
    rectanglefText.Height = NumDisplayLines * HeightFont;
```

Елемент **RichTextBox** має властивість **WordWrap**, при встановленні у значення **true**, текст буде автоматично переноситися до нового рядку. Проте, розділений на рядки текст може вийти за межі визначеної для нього області **rectanglefText**. Щоб цього не сталося, потрібно створити екземпляр **stringformat** класу **StringFormat**, для якого необхідно встановити значення **Trimming**:

```
if (rtbText.WordWrap)
    stringformat.Trimming = StringTrimming.Word;
else {
    stringformat.Trimming = StringTrimming.EllipsisCharacter;
    stringformat.FormatFlags |= StringFormatFlags.NoWrap;
}
//Друк обраних сторінок починається з першої стартової сторінки
while ((PageNumber<StartPage)&&(stringPrintText.Length>0)) {
    if (rtbText.WordWrap)
        //Вимірюються текстові змінні, формується друк та
        //повертається число символів та кількість рядків:
```

```

        graph.MeasureString(stringPrintText, font,
rectanglefText.Size, stringformat, out NumberSymbols,
out NumberLines);
        else
            NumberSymbols = SymbolsInLines(stringPrintText,
NumDisplayLines);
            stringPrintText = stringPrintText.Substring(NumberSymbols);
            //Збільшується кількість сторінок
            PageNumber++;
        }
        //Якщо довжина рядку stringPrintText дорівнює нулю (текст
відсутній), то потрібно зупинити друк
        if (stringPrintText.Length == 0) {
            e.Cancel = true;
            return;
        }
        //Виводиться текст для друку - stringPrintText, шрифтом font,
//кисть чорного кольору - Brushes.Black, область друку -
//rectanglefText, передається рядок додаткового форматування:
stringformat graph.DrawString(stringPrintText, font,
Brushes.Black, rectanglefText, stringformat);
        //Отримується текст для наступної сторінки:
        if (rtbText.WordWrap)
            graph.MeasureString(stringPrintText, font,
rectanglefText.Size, stringformat, out NumberSymbols, out
NumberLines);
        else
            NumberSymbols = SymbolsInLines(stringPrintText,
NumDisplayLines);
            stringPrintText = stringPrintText.Substring(NumberSymbols);
            //Очищується об'єкт stringformat для формування полів:
            stringformat = new StringFormat();
            //Додається номер до кожної сторінки
            stringformat.Alignment = StringAlignment.Far;
            graph.DrawString("Сторінка " + PageNumber, font, Brushes.Black,
rectanglefFull, stringformat);
            PageNumber++;
            //Знову перевіряється, чи є текст для друку та номер сторінки:
            e.HasMorePages = (stringPrintText.Length > 0) && (PageNumber <
StartPage + NumPages);
            //Для друку з вікна попереднього перегляду потрібно знову
//ініціалізувати змінні
            if (!e.HasMorePages) {
                stringPrintText = rtbText.Text;
                StartPage = 1;
                NumPages = printDialog1.PrinterSettings.MaximumPage;
                PageNumber = 1;
            }
}

```

Метод **MeasureString** повертає число символів, що виводяться. Для обліку символів, що не потрапляють до області **rectanglefText**, додається метод:

```

int SymbolsInLines(string stringPrintText, int NumLines) {
    int index = 0;
    for (int i = 0; i < NumLines; i++) {
        index = 1 + stringPrintText.IndexOf('\n', index);
        if (index == 0)
            return stringPrintText.Length;
    }
    return index;
}

```

Надалі створюються обробники подій пунктів **mnuPageSetup** та **mnuPrintPreview** ГОЛОВНОГО МЕНЮ:

```

private void mnuPageSetup_Click(object sender,
System.EventArgs e) {
    pageSetupDialog1.ShowDialog();
}
private void mnuPrintPreview_Click(object sender,
System.EventArgs e) {
    printDocument1.DocumentName = Text;
    stringPrintText = rtbText.Text;
    StartPage = 1;
    NumPages = printDialog1.PrinterSettings.MaximumPage;
    PageNumber = 1;
    printPreviewDialog1.ShowDialog();
}

```

В обробнику пункту меню **mnuPrint** необхідно визначити діапазон сторінок – усі сторінки, рядок або виділена область:

```

private void mnuPrint_Click(object sender, System.EventArgs e) {
    printDialog1.AllowSelection = rtbText.SelectionLength > 0;
    if (printDialog1.ShowDialog() == DialogResult.OK) {
        printDocument1.DocumentName = Text;
        //Визначається діапазон сторінок для друку:
        switch (printDialog1.PrinterSettings.PrintRange) {
            //Вибрано усі сторінки:
            case PrintRange.AllPages:
                stringPrintText = rtbText.Text;
                StartPage = 1;
                NumPages = printDialog1.PrinterSettings.MaximumPage;
                break;
            //Обрано виділену область:
            case PrintRange.Selection:
                stringPrintText = rtbText.SelectedText;
                StartPage = 1;
                NumPages = printDialog1.PrinterSettings.MaximumPage;
                break;
            //Вибрано ряд сторінок:
            case PrintRange.SomePages:
                stringPrintText = rtbText.Text;
                StartPage = printDialog1.PrinterSettings.FromPage;
                NumPages = printDialog1.PrinterSettings.ToPage -
StartPage + 1;

```

```

        break;
    }
    PageNumber = 1;
    //Викликається вбудований метод для початку друку:
    printDocument1.Print();
}
}

```

Після запуску застосунку на виконання, зміни, які зроблені в діалоговому вікні "Параметри сторінки" зберігаються та у вікні попереднього перегляду відображено підготовлену для друку сторінку. З вікна попереднього перегляду можна відразу перейти до друку. При виборі пункту друку доступно діапазон сторінок або друк виділеного фрагменту.

2.2 Створення елементів керування за допомогою .NET Framework

2.2.1 Застосування GDI+

Windows – операційна система з графічним інтерфейсом. Користувачі працюють з вікнами, які в графічній формі представляють дані та функціональність застосунків. .NET Framework надає безліч графічних елементів керування та підтримує різноманітні методи створення програм з графічним інтерфейсом. Але іноді потрібно формувати та відображати власне графічне подання, а також генерувати елементи керування нестандартного вигляду. Щоб повною мірою скористатися перевагами Windows, необхідно засвоїти можливості інтерфейсу графічних пристроїв (Graphic Device Interface, GDI). У .NET Framework повністю реалізовано керовану версію цього інтерфейсу під назвою GDI+. Терміном GDI+ у .NET Framework позначають керовану реалізацію GDI, яка застосовується для відображення графічної інформації на дисплеї комп'ютера. Доступ до цього інтерфейсу надається через шість просторів імен, де реалізовано різні види функцій GDI+.

Простір імен **System.Drawing**

Простір імен **System.Drawing** надає надзвичайно широкі можливості. Детальне дослідження компонентів цього простору імен не є метою даних методичних вказівок, але потрібно розглянути застосування його класів та методів (табл. 2.4).

Таблиця 2.4 – Простір імен **System.Drawing**

Ім'я	Що містить
System.Drawing	Основні класи, які беруть участь у відображенні графіки на екрані. Це ключовий простір імен для програмування графічного інтерфейсу
System.Drawing.Design	Класи, які надають інтерфейсу додаткові можливості періоду розробки
System.Drawing.2D	Класи, що відображають складні графічні ефекти
System.Drawing.Imaging	Класи, що надають додаткові можливості по маніпулюванню графічними файлами
System.Drawing.Printing	Класи для друку
System.Drawing.Text	Класи, що надають додаткові можливості з керування шрифтами

Ключову роль у формуванні та відображенні графіки грає клас **Graphics** з простору імен **System.Drawing**. Об'єкт **Graphics** представляє область для малювання, розташовану на поверхні графічного елемента – форми, елемента керування або об'єкта **Image**. З усіма цими сутностями пов'язано об'єкт **Graphics**, який дозволяє малювати на їх поверхні та відображає всі графічні елементи, які на неї розташовано.

Об'єкт **Graphics**

Оскільки об'єкт **Graphics** повинен бути пов'язаний з графічним елементом, його не можна створити, викликавши конструктор цього класу безпосередньо. Замість цього потрібно викликати конструктор об'єкту **Graphics** з графічного елемента. Будь-який клас-нащадок **Control** (включно **Form**) підтримує метод **CreateGraphics**. Останній повертає посилання до об'єкту **Graphics**, пов'язане з даним елементом керування. Наступний приклад демонструє отримання доступу до об'єкту **Graphics** форми **myForm**:

```
System.Drawing.Graphics myGraphics;
myGraphics = myForm.CreateGraphics();
```

У результаті виконання цього коду створюється об'єкт **Graphics**, яким можна користуватися для відображення графіки на поверхні форми.

При роботі із зображеннями метод **Graphics.FromImage** дозволяє створити об'єкт **Graphics**, пов'язаний із заданим об'єктом **Image**. Це статичний метод, тому для його виклику посилання на об'єкт **Graphics** не потрібно. Об'єктом **Image** вважається будь-який об'єкт-нащадок класу **Image**, наприклад **Bitmap**. Приклад створення об'єкту **Bitmap** та отримання пов'язаного з ним об'єкту **Graphics**:

```
Bitmap myImage = new Bitmap("C:\\myImage.bmp");
System.Drawing.Graphics myGraphics;
myGraphics = Graphics.FromImage(myImage);
```

Навіть невидимий об'єкт **Image** дозволяє створити об'єкт **Graphics** та маніпулювати ним.

Формування зображення відбувається в області, заданої межами елемента керування. Тут діє двовимірна система координат, кожна точка якої задана парою значень **X** та **Y**. За замовчуванням початкову точку системи координат **(0, 0)** кожного елемента розташовано в його верхньому лівому куті, а координати задаються у пікселях. Простір імен **System.Drawing** містить ряд структур (табл. 2.5), які описують точку або область у даній системі координат.

Таблиця 2.5 – Структури опису координат точок та фігур

Ім'я	Опис
Point	Представляє точку з координатами X та Y, заданими значеннями типу Integer (int)
PointF	Представляє точку з координатами X та Y, заданими значеннями типу Single (float)
Size	Представляє прямокутну область, розміри якої задані парою значень Height та Width типу Integer (int)
SizeF	Представляє прямокутну область, розміри якої задані парою значень Height та Width типу Single (float)
Rectangle	Представляє прямокутну область, межі якої задані значеннями Top, Bottom, Left та Right типу Integer (int)
RectangleF	Представляє прямокутну область, межі якої задані значеннями Top, Bottom, Left та Right типу Single (float)

Структури **Point**, **Size** та **Rectangle** цілого типу дозволяється явно перетворювати до відповідних структур дійсного типу з попереднім перетворенням кожної координати з цілого типу до типу з плаваючою крапкою, наприклад:

```
Point myPoint;
PointF myPointF = new PointF(13.5F, 33.21F);
myPoint = new Point((int)myPointF.X, (int)myPointF.Y);
```

Особливістю структур **Size** та **Rectangle** є те, що обидві вони представляють прямокутну частину клієнтської області, однак **Size** визначає тільки розмір але не положення прямокутника, тоді як структура **Rectangle** задає реальний стан прямокутника у клієнтській області. Для створення **Rectangle** потрібні дві структури: **Size** та **Point**, які вказують положення верхнього лівого кута прямокутника:

```
Point myOrigin = new Point(10, 10);
Size mySize = new Size(20, 20);
Rectangle myRectangle = new Rectangle(myOrigin, mySize);
```

Об'єкт **Graphics** інкапсулює безліч методів для малювання на дисплеї простих та складних фігур. Існує два основні різновиди цих методів. До першого відносяться методи, імена яких починаються з «**Draw**» (табл. 2.6). Ці методи використовуються для малювання прямих, дуг та контурів фігур.

Таблиця 2.6 – Методи для малювання контурів фігур

Ім'я	Опис
DrawArc	Малює дугу, що представляє частину еліпсу
DrawBezier	Малює криву Безьє
DrawBeziers	Малює кілька кривих Безьє
DrawClosedCurve	Проводить через кілька точок замкнену криву
DrawCurve	Проводить через кілька точок незамкнену криву
DrawEllipse	Малює еліпс, вписаний до прямокутника
DrawLine	Поєднує дві точки лінією
DrawPath	Малює складну фігуру GraphicsPath
DrawPie	Малює сектор (кульовий)
DrawPolygon	Малює багатокутник за заданою кількістю точок
DrawRectangle	Малює прямокутник
Draw Rectangles	Малює кілька прямокутників

Інший різновид методів, імена яких починаються з «**Fill**» (табл. 2.7), призначено для малювання заповнених фігур-прямокутників, еліпсів та багатокутників.

Таблиця 2.7 – Методи для малювання заповнених фігур

Ім'я	Опис
FillClosedCurve	Малює замкнуту криву, задану масивом точок та зафарбовує отриману фігуру
FillEllipse	Малює зафарбований еліпс
FillPath	Малює та зафарбовує складний об'єкт GraphicsPath
FillPie	Малює зафарбований сектор
FillPolygon	Малює багатокутник, заданий масивом точок та зафарбовує його
FillRectangle	Малює зафарбований прямокутник
FillRectangles	Малює кілька зафарбованих прямокутників
FillRegion	Малює та зафарбовує складний об'єкт Region

Ці методи приймають різні комбінації параметрів, які визначають розміри та положення відображуваних фігур. Крім того, їм потрібен об'єкт, що формує зображення фігури: для контурних фігур – **Pen**, а для зафарбованих – **Brush**.

Об'єкти Color, Brush та Pen

Об'єкти **Color**, **Brush** та **Pen** формують вигляд графічного елементу. Об'єкт **Brush** використовують для відображення зафарбованих фігур, **Pen** – для відображення прямих і дуг, а об'єкт **Color** задає колір фігури.

Color. Структура **Color** розташовується в просторі імен **System.Drawing** і представляє певний колір. Кожен колір визначає чотири значення: **Alpha**, яке надає прозорість, і три значення – **Red**, **Green** та **Blue** з діапазону 0–255. Щоб створити новий колір, необхідно передати ці значення методу **Color.FromArgb**:

```
Color myColor;
myColor = Color.FromArgb(128, 255, 12, 43);
```

Якщо ефект прозорості не використовується, параметр **Alpha** можна пропустити, вказавши лише значення **Red**, **Green** та **Blue**:

```
Color myColor;
myColor = Color.FromArgb(255, 12, 43);
```

Крім того, безліч стандартних кольорів є в .NET Framework у вигляді іменованих констант:

```
Color myColor;
myColor = Color.Tomato;
```

Brush. Об'єкт **Brush** представляє кисть і служить для малювання зафарбованих фігур. Усі різновиди кистей є наслідуванням абстрактного класу **Brush** та дозволяють створювати зафарбовані фігури різного виду. Типи кистей та простори імен, в яких розташовано відповідні об'єкти, описано у таблиці 2.8.

Таблиця 2.8 – Типи кистей

Тип	Простір імен	Опис
SolidBrush	System.Drawing	Суцільна одноколірна кисть
TextureBrush	System.Drawing	Кисть для заливки замкнутого контуру зображенням
HatchBrush	System.Drawing.Drawing2D	Кисть для візерункової заливки
LinearGradientBrush	System.Drawing.Drawing2D	Кисть для заливки двоколірним градієнтом
PathGradientBrush	System.Drawing.Drawing2D	Кисть для складної візерункової заливки

Для створення об'єкту **SolidBrush** достатньо вказати колір, наприклад:

```
SolidBrush myBrush = new SolidBrush(Color.PapayaWhip);
```

Конструктори інших кистей складніші та вимагають додаткових параметрів. Наприклад, для створення кисті типу **TextureBrush** необхідно об'єкт **Image**, а для **LinearGradientBrush** – два кольори та декілька інших параметрів, в залежності від типу обраного конструктора.

Pen. Об'єкти **Pen** представляють пір'я, вони призначені для малювання прямих та дуг, а також для застосування до контурних фігур різних графічних ефектів.

Існує тільки один клас **Pen**, але він є (sealed). Створити об'єкт класу **Pen** нескладно, достатньо вказати його колір:

```
Pen myPen = new Pen(Color.BlanchedAlmond);
```

В результаті виконання попереднього коду створюється перо, ширина якого за замовчуванням дорівнює **1** пікселю. Ширину задають у конструкторі об'єкту **Pen** наступним чином:

```
Pen myPen = new Pen(Color.Lime, 4);
```

Цей код призначає ширину пера, яка дорівнює чотирьом пікселям. Перо можна створити і на основі наявної кисті, вид такого пір'я буде узгоджено зі стилем інтерфейсу. Це особливо зручно при використанні складних тіней та інших ефектів. Наступний приклад демонструє створення пера на основі наявної кисті **myBrush**:

```
Pen myPen = new Pen(myBrush);
```

При створенні пера на основі кисті також дозволяється задавати її ширину.

Системні кисті, пір'я та кольори

При розробці користувацького інтерфейсу іноді нестандартні компоненти доводиться оформляти в тому ж стилі, що й системні. Для цього .NET Framework надає доступ до системних кольорів через клас **SystemColors**, який містить набір статичних компонент, які представляють активні системні кольори. При розробці нестандартних компонентів користувацького інтерфейсу можна призначити їм системні кольори, щоб у період виконання вони відображалися з використанням активної системної палітри. Отримати доступ до системних кольорів можна так:

```
Color myColor = SystemColors.HighlightText;
```

Поряд з **SystemColors**, у .NET Framework також є класи **SystemPens** та **SystemBrushes**, які надають доступ до пір'я та кистей, що застосовуються за замовчуванням. Ці кисті працюють так само, як будь-які інші, представлені класами **Pen** або **SolidBrush**.

Малювання простих фігур

Клас **Graphics** підтримує ряд методів, що дозволяють малювати прості фігури.

Будь-яким методам для малювання контурних фігур необхідно дійсний об'єкт **Pen**, а методам, які малюють зафарбовані фігури – дійсний об'єкт **Brush**. Крім того, при виклику цих методів варто передати будь-які необхідні їм об'єкти. Так, наприклад, можна намалювати прямокутник засобами методу **DrawRectangle**:

```
//Створення об'єкту Rectangle:
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
//Створення об'єкту Graphics, пов'язаного з формою:
Graphics g = this.CreateGraphics();
//Малювання прямокутника системним пером:
g.DrawRectangle(SystemPens, ControlDark, myRectangle);
//Звільнення ресурсів, зайнятих об'єктом Graphics:
g.Dispose();
```

Закінчивши роботу з об'єктом **Graphics**, необхідно викликати обов'язково метод **Dispose**. В іншому випадку швидкодія програми може знизитися, тому що цей об'єкт використовує багато системних ресурсів. Аналогічні дії виконуються з будь-якими створеними об'єктами **Pen** та **Brush**. Наступний приклад коду показує, як намалювати зафарбований еліпс та коректно звільнити об'єкти **Brush** та **Graphics**:

```
SolidBrush myBrush = new SolidBrush(Color.MintCream);
Graphics g = this.CreateGraphics();
//Створюється еліпс, вписаний до прямокутника:
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
g.FillEllipse(myBrush, myRectangle);
//Звільнення ресурсів об'єктів Graphics та Brush:
g.Dispose();
myBrush.Dispose();
```

Як намалювати просту фігуру:

- створіть об'єкт **Graphics**, що представляє область, в якій потрібно намалювати фігуру;
- створіть необхідні допоміжні об'єкти. До них відносяться об'єкти, що задають координати та розміри фігур, наприклад **Point** або **Rectangle**, а також об'єкти **Pen** (для малювання контурних фігур) та **Brush** (для зафарбованих фігур);
- викличте відповідний метод об'єкту **Graphics**;
- звільніть усі ресурси, зайняті створеними об'єктами **Pen** або **Brush**;
- звільніть ресурси об'єкту **Graphics**.

Відображення тексту

Метод **DrawString** об'єкту **Graphics** дозволяє відобразити на дисплеї текстовий рядок як графічний елемент, намалювавши його в певному місці дисплею заданим пензлем, крім об'єкту **Brush**. Цим методом необхідно передати координати верхнього лівого кута відображуваного тексту (у вигляді структури **PointF**). Нижче

наведено приклад коду, який відображає рядок за допомогою методу **Graphics.DrawString**:

```
//Використання системної кисті класу SystemBrush:
Graphics g = this.CreateGraphics();
String myString = "Hello World";
Font myFont = new Font("Times New Roman", 36, FontStyle,
Regular);
//Останні два параметри представляють координати рядка:
g.DrawString(myString, myFont, SystemBrushes.Highlight, 0, 0);
//Обов'язкове звільнення об'єкту Graphics:
g.Dispose();
```

Відображення тексту як графічного елементу:

- при необхідності створіть об'єкти **Font** та **Brush**, які визначають вид об'єкту **String**;
- отримайте посилання на об'єкт **Graphics**, пов'язане з областю, в якій повинно бути відображено текст;
- викличте метод **Graphics.DrawString**, передавши об'єкти **String**, **Font**, **Brush** та координати верхнього лівого кута області, в якій буде відображено текст;
- звільніть об'єкт **Graphics**, викликавши метод **Graphics.Dispose**.

Малювання складних фігур

Час від часу доводиться малювати фігури більш складні, ніж прямокутники, еліпси або багатокутники. У об'єктів простих фігур є вбудовані методи, які полегшують їх малювання, тоді як малювання складних фігур вимагає додаткової підготовки. Ключовим для візуалізації складних фігур є об'єкт **GraphicsPath** з простору імен **System.Drawing.Drawing2D**. Він описує довільний замкнутий контур або набір фігур. Так, можна створити об'єкт **GraphicsPath**, утворений еліпсами, прямокутниками та іншими простими об'єктами, які представляють фігуру неправильної форми.

Створення об'єкту **GraphicsPath**

Об'єкт **GraphicsPath** створюють викликом однієї з версій його конструктора. Найпростіша версія конструктора об'єкту **GraphicsPath** не містить параметрів:

```
GraphicsPath myPath = new Drawing2D.GraphicsPath();
```

Крім того, об'єкт **GraphicsPath** допустимо задавати у вигляді масиву точок та значень типу **Byte**. Точки визначають координати контуру, а значення **Byte** – тип ліній, якими ці точки з'єднуються. Перетворивши подібний масив з переліченого типу **System.Drawing.Drawing2D.PathPointType**, з'являється можливість створити більш зрозумілий і зручний код. Приклад коду, що створює дуже простий об'єкт **GraphicsPath**:

```
using System.Drawing.Drawing2D;
GraphicsPath myPath = new GraphicsPath(new Point[]
{new Point(1,1), new Point(32,54), new Point(33,5)}, new byte[]
{ (Byte)PathPointType.Start, (Byte)PathPointType.Line,
(Byte)PathPointType.Bezier});
```

До створеного об'єкту **GraphicsPath** можна додавати фігури. Фігура – це замкнутий контур, який може бути простим (як еліпс або прямокутник) або складним (як довільні криві та контури символів).

Методи класу **GraphicsPath** дозволяють додавати до контуру нові фігури (таблиця 2.9).

Таблиця 2.9 – Методи додавання фігур до **GraphicsPath**

Ім'я	Що додає
AddClosedCurve	Замкнута крива, задана масивом точок
AddEllipse	Еліпс
AddPath	Поставлений екземпляр об'єкту GraphicsPath
AddPie	Коло
AddPolygon	Багатокутник, заданий масивом точок
AddRectangle	Прямокутник
AddRectangles	Масив прямокутників
AddString	Графічна уява рядку символів із заданим накресленням

Дозволяється не тільки безпосередньо додавати фігури до об'єкту **GraphicsPath**, але й складати контури, додаючи прямі, криві та дуги. Щоб почати новий контур, необхідно викликати метод **GraphicsPath.StartFigure**. Далі, використовуючи методи класу **GraphicsPath**, можна додавати до контуру нові фігури. По закінченню конструювання фігури потрібно викликати метод **GraphicsPath.CloseFigure**, щоб замкнути її контур – остання точка фігури автоматично з'єднується з першою. Приклад:

```
GraphicsPath myPath = new Drawing2D.GraphicsPath();
myPath.StartFigure();
```



```
//Додавання елементів контуру
myPath.CloseFigure();
```

Якщо викликати метод **StartFigure** два рази поспіль, не викликавши між ними метод **CloseFigure**, то фігура, створена першим викликом **StartFigure**, залишиться незамкненою.

При відображенні в період виконання будь-яка незамкнута фігура автоматично замикається додаванням прямої, яка з'єднає першу та останню точку контуру цієї фігури.

У таблиці 2.10 наведено методи об'єкту **GraphicsPath**, які дозволяють додавати нові елементи контуру фігури.

Таблиця 2.10 – Методи для додавання елементів контуру

Ім'я	Що додає
AddArc	Дуга
AddBezier	Крива Без'є
AddBeziers	Набір кривих Без'є
AddCurve	Крива, задана масивом точок
AddLine	Пряма
AddLines	Набір з'єднаних прямих

Як намалювати складну фігуру:

- отримайте посилання на об'єкт **Graphics**, пов'язане з областю малювання, в якій потрібно намалювати фігуру;
- створіть новий екземпляр класу **GraphicsPath**;
- додайте до нього фігури за допомогою методів класу **GraphicsPath**;
- викличте метод **Graphics.DrawPath**, щоб намалювати контур, або **Graphics.FillPath**, щоб намалювати зафарбовану фігуру, яка задана об'єктом **GraphicsPath**;
- звільніть об'єкт **Graphics**.

2.2.2 Створення елементів керування

В основі програмування із застосуванням Windows Forms лежить використання елементів керування. Вони інкапсулюють окремі функції програми та є їх графічним поданням для користувача. .NET Framework надає в розпорядження розробника ряд інструментів та методів для створення власних елементів керування. Елементи керування .NET Framework – це спеціалізовані класи, які містять код,

що формує їх графічний інтерфейс. Існує декілька джерел елементів керування, які застосовуються у застосунках. По-перше, це стандартні елементи керування Visual Studio .NET, розроблені Microsoft. Вони забезпечують широкі можливості та дозволяють розробляти клієнтські програми з багатою функціональністю. По-друге, у застосунку припустимо використовувати елементи керування від сторонніх розробників, а також існуючі елементи керування ActiveX. Елементи керування від сторонніх розробників підтримують можливості, відсутні в стандартних компонентах з .NET Framework. Нарешті, якщо функції, необхідні застосунку, не реалізовано ні в стандартних елементах керування .NET Framework, ні в елементах керування сторонніх розробників, то дозволяється створювати власні елементи керування.

Принципи створення елементів керування

Усі елементи керування .NET Framework є прямим або непрямым наслідуванням класу **Control** та успадковують від нього базовий набір низькорівневої функціональності, необхідної всім елементам керування. Наприклад, **Control** надає код для обробки даних, введених користувачем за допомогою клавіатури і миші, а також код, який взаємодіє з ОС. Крім того, класи елементів керування отримують від свого прашура набір властивостей, методів і подій, загальних для всіх елементів керування. Однак, базовий клас не передає своїм нащадкам ні специфічних функцій, ні коду, що формує інтерфейс елемента керування.

Основних підходів до створення елементів керування три:

- а) наслідування від існуючих елементів керування;
- б) об'єднання стандартних елементів керування у групи;
- в) створення нестандартних елементів керування «з нуля».

Наслідування від існуючих елементів керування

Простіше за все створити новий елемент керування, оголосивши його на основі існуючого. Створений таким чином елемент керування не тільки зберігає всю функціональність базового компоненту, але і дозволяє доповнити її новими можливостями.

Крім функціональності, похідний елемент керування наслідує від свого нащадка вигляд інтерфейсу. Наприклад, елемент керування,

похідний від **Button**, зовні не відрізняється від звичайної кнопки. Це звільняє від створення інтерфейсу для похідних елементів керування, але при необхідності це можна зробити.

Більшість елементів керування Windows Forms, якщо вони не позначені ключовим словом **NotInheritable** (sealed), можуть бути базовими класами для нових елементів керування. Наприклад, це дозволяє створити новий елемент керування **TextBox** із вбудованим кодом для перевірки значення поля або **PictureBox**, підтримуючий встановлення користувачем фільтрів для відображаються в ньому зображень. Наслідування також дозволяє створювати елементи керування, функціонально ідентичні базовому елементу керування, але різні за зовнішнім виглядом. Приклад – нестандартна кнопка, яка відрізняється від традиційної прямокутної кнопки круглою формою.

Оголошення нових елементів керування за допомогою наслідування вважається найпростішим способом їх створення та вимагає мінімальних витрат часу. Даний спосіб зручний і в тих випадках, коли потрібно змінити вигляд елементу керування, зберігши його функціональність. Якщо ж потрібно створити елемент керування із кардинально іншою функціональністю, то не слід оголошувати його на основі існуючого.

Наслідування від класу **UserControl**

Іноді функціональності одного елементу керування недостатньо для вирішення поставлених завдань. Припустимо, що потрібен елемент керування, який підтримує зв'язування з джерелом даних та відображає ім'я, прізвище і номер телефону в елементах керування **TextBox**. Можна додати до форми відповідний код, але раціональніше створити новий елемент керування, який містить кілька текстових полів. Це особливо зручно, коли подібні можливості потрібні декільком застосункам. Елементи керування, утворені об'єднаними до групи стандартними елементами керування Windows Forms називаються призначеними для користувача або складовими.

Складові елементи керування оголошують на основі класу **UserControl**, що надає базову функціональність, яку можна розширити шляхом додавання властивостей, методів, подій та інших елементів керування. Дизайнер складових елементів керування дозволяє розміщувати на їх поверхні стандартні елементи керування

Windows Forms та додавати код, який реалізує специфічні можливості. Таким чином, користувач може створювати нові елементи керування, об'єднуючи стандартні елементи керування Windows Forms до логічних груп.

Наслідкування від класу **Control**

Якщо потрібен більш складний інтерфейс або можливості, які не вдається реалізувати із застосуванням складових та похідних елементів керування, звертаються до створення формованих елементів керування. Формовані елементи є прямими нащадками **Control** – базового класу всіх елементів керування. Клас **Control** передає «у спадщину» великий об'єм функціональності, загальну для всіх елементів керування, наприклад, підтримку взаємодії з мишею та спільні події (наприклад, **Click**), а також полегшують роботу з елементами керування властивості, такі як **Font**, **ForeColor**, **BackColor**, **Visible** та інші.

Проте, клас **Control** не надає похідним від нього елементам керування ніяких спеціальних можливостей. Всю логіку, необхідну для вирішення специфічних завдань даного елемента керування, повинен реалізувати розробник. Хоча клас **Control** надає значну частину властивостей, які визначають обличчя інтерфейсу, розробнику доведеться самостійно створити весь код графічного інтерфейсу елемента керування.

Створення формованих елементів керування потребує найбільше часу і зусиль від програміста, оскільки на його плечі лягає не тільки реалізація нестандартних можливостей даного елемента керування, але й створення його графічного представлення, що може відняти багато часу. Формувати елементи керування самостійно потрібно, тільки якщо потрібна складна форма або функціональність, яку не надає жоден інший тип елементів керування.

Додавання членів до елементів керування

Процедура додавання властивостей, методів, полів і подій до елементів керування не відрізняється від додавання членів до звичайного класу. Члени класів елементів керування дозволяється оголошувати з будь-яким підходящим рівнем доступу, щоб зробити їх доступними формі, на якій розташовано елемент керування.

Відкриті властивості, додані до елементів керування в період розробки, автоматично відображаються у вікні **Properties**, де їх може відредагувати і налаштувати будь-який користувач. Якщо потрібно сховати властивість з вікна **Properties**, то необхідно додати до його оголошення атрибут **Browsable** та встановити його у **false**.

Приклад:

```
<System.ComponentModel.Browsable (false)> public int
StockNumber {
    //Реалізація властивості
}
```

Створення похідних елементів керування

Похідний елемент керування містить всю функціональність свого базового класу, а також ряд додаткових можливостей. Щоб створити похідний елемент керування, необхідно вказати для нього базовий клас, наприклад, клас стандартного елемента керування Windows Forms. Новий елемент керування, створений таким чином, збереже зовнішній вигляд та поведінку успадкованого елемента. Нижче наведено приклад створення елемента керування, похідного від **Button**:

```
public class myButton:System.Windows.Forms.Button {
    //Реалізація властивості
}
```

Реалізація нових можливостей в похідних елементах керування

Як правило, до похідних елементів керування вдаються, коли потрібно реалізувати в них додаткові функції, які відсутні в стандартних елементах керування Windows Forms. Для цього до похідних елементів керування додають новий код або перевизначають члени, успадковані від базового класу. Припустимо, що розробнику потрібно створити текстове поле, куди користувач зможе вводити лише числа. Для цього необхідно перевизначити метод **OnKeyPress** наступним чином:

```
public class NumberBox:System.Windows.Forms.TextBox {
    protected override void OnKeyPress(KeyPressEventArgs e) {
        if (char.IsNumber(e.KeyChar) == false)
            e.Handled = true;
    }
}
```

Модифікація зовнішнього вигляду існуючих елементів керування

Інша причина, за якій створюють похідні елементи керування – їх нестандартний зовнішній вигляд. Щоб змінити стиль оформлення існуючого елементу керування, потрібно перевизначити метод **OnPaint**, помістивши до нього власний код. Наприклад, щоб змінити форму елементу керування, встановлюється відповідним чином його властивість **Region** у методі **OnPaint**. Клас **Region**, подібно **GraphicsPath**, описує область екрану заданої форми, який отримують з класу **GraphicsPath**. Так можна згенерувати клас **GraphicsPath**, що представляє форму вашого елементу керування, створити на його основі об'єкт **Region** та помістити у відповідну властивість елементу керування. Цей прийом ілюструється на прикладі створення кнопки у формі тексту:

```
public class WowButton: System.Windows.Forms.Button {
    protected override void OnPaint(PaintEventArgs pe) {
        System.Drawing.Drawing2D.GraphicsPath myPath =
new System.Drawing.Drawing2D.GraphicsPath();
        //Наступна команда встановлює FontFamily та FontStyle
        //елементу керування так, щоб об'єкт GraphicsPath
        //представляв контур рядка символів розміром 72 пункти.
        //У C# треба явно перетворити значення Font.Style до цілого
        myPath.AddString("Wow! ", Font.FontFamily, (int)Font.Style,
72, new PointF(0, 0), StringFormat.GenericDefault);
        //Створення на основі GraphicsPath нового об'єкту Region:
        Region myRegion = new Region(myPath);
        //Помістити новий об'єкт Region до однойменної властивості
        //елементу керування:
        this.Region = myRegion;
    }
}
```

Однак, **WowButton** – це не прямокутна кнопка з прозорим фоном, що відображає великі літери, вона дійсно має форму букв, які становлять слово «Wow!». Щоб її натиснути, необхідно потрапити до області, яка обмежена контуром букв. Подібним чином змінюють форму звичайних елементів керування, щоб отримати елементи користувацького інтерфейсу з нестандартним виглядом. Адже, промальовування деяких елементів керування, таких як **TextBox**, виконується не самим елементом керування, а формою, на якій його розташовано. Такі елементи керування ніколи не генерує подія **Paint**, тому що код візуалізації у обробнику, викликаний не буде.

Створення похідного елементу керування:

- оголосіть новий клас елементу керування на основі існуючого класу елементу керування;
- реалізуйте всі необхідні нестандартні можливості та додайте код, що формує інтерфейс похідного елементу керування.

Створення складових елементів керування

Користувацькі або складові елементи керування об'єднують нестандартні можливості з функціональністю стандартних елементів керування Windows Forms, дозволяючи швидко створювати нові елементи керування. Користувацький елемент керування складається з одного або декількох стандартних елементів керування, об'єднаних до групи. Дизайнер **UserControl** призначений для додавання до складових елементів керування дочірніх елементів керування.

Розробник може написати власний код для обробки подій, керуючись вкладеними елементами керування. Припустимо, що потрібно створити користувацький елемент керування, який складається з двох текстових полів та напису. При зміні значень будь-якого з текстових полів цей елемент керування має автоматично виконувати арифметичне додавання одного до іншого та відображати суму в написі. Подібну функціональність вдається реалізувати, перевизначивши методи **OnKeyPress** вкладених текстових полів наступним чином:

```
//Для події KeyPress елементу керування TextBox2
//потрібно створити аналогічний обробник:
protected override void OnKeyPress(object sender,
KeyPressEventArgs e) {
    //Перевірка, чи є натиснута клавіша цифровою:
    if (char.IsNumber(e.KeyChar) == false)
        e.Handled = true;
    Label1.Text = (int.Parse(TextBox1.Text) +
int.Parse(TextBox2.Text)).ToString();
}
```

Вкладені компоненти складових елементів керування вважаються закритими. Тому жоден розробник, який буде працювати з вашим елементом керування, не зможе змінити такі властивості його вкладених елементів керування, як колір, форма і розмір. Щоб дозволити іншим розробникам змінювати властивості вкладених елементів керування, потрібно зробити їх доступними через властивості об'єкту складового елементу керування.

На прикладі складового елемента керування з вкладеною кнопкою (**Button1**) для надання доступу до властивості **BackColor** об'єкту **Button1**, створюється для нього властивість – оболонка об'єкту елемента керування:

```
public color ButtonColor {
    get {
        return Button1.BackColor;
    }
    set {
        Button1.BackColor = value;
    }
}
```

Щоб надати доступ до вкладеного елемента керування, змінюється його властивість **Modifiers**. Це можливо тільки через вікно **Properties** під час розробки, оскільки під час виконання воно не існує. Властивість **Modifiers** дозволяє задати будь-який рівень доступу, який застосовується до відповідного вкладеному елементу керування.

Як створити складовий елемент керування:

- оголосіть клас, похідний від **UserControl**;
- викличте дизайнер **UserControl** та додайте за його допомогою вкладені елементи керування Windows Forms, після чого відповідним чином налаштуйте їх;
- при необхідності зробіть доступними властивості вкладених елементів керування;
- реалізуйте нестандартні можливості, необхідні складеному елементу керування.

Створення формованих елементів керування

Формовані елементи керування надають максимум можливостей по налаштуванню та керуванню своєю функціональністю, але їх розробка займає найбільше часу. Оскільки базовий клас **Control** не надає своїм нащадкам навіть базового графічного інтерфейсу, розробнику доведеться самостійно писати весь код, що формує образ такого елемента керування.

Процес візуалізації елемента керування в клієнтській області форми називається промальовуванням. Отримавши команду на промальовування, елемент керування генерує подію **Paint**, при

цьому виконуються всі обробники події **Paint**. У класі **Control** обробником за замовчуванням для події **Paint** є метод **OnPaint**.

Метод **OnPaint** приймає єдиний аргумент – екземпляр класу **PaintEventArgs**, що містить інформацію клієнтської області елемента керування. Варто звернути увагу на два члени цього класу – **Graphics** та **ClipRectangle**.

Graphics – це об'єкт типу **Graphics**, що представляє клієнтську область елемента керування. Посилання на нього необхідне для формування зображення елемента керування. **ClipRectangle** – прямокутник, який надає доступну клієнтську область елемента керування. При першому відображенні елемента керування, **ClipRectangle** обмежує зайняту їм область. Якщо вона перекривається з іншими елементами керування, то цей елемент може виявитися частково або повністю прихованим. При повторному його промальовуванні **ClipRectangle** представляє тільки видиму область, яка повинна бути перемальована. Саме тому не рекомендується визначати розміри елемента керування за об'єктом **ClipRectangle**, а для цієї мети слід застосовувати властивість **Size**.

За замовчуванням координати мають відлік від верхнього лівого кута елемента керування, в якому розташовується умовна точка початку координат **(0, 0)**, вони обчислюються в пікселях. Наступний приклад демонструє простий метод **OnPaint**, що виконує промальовування нестандартного елемента керування у вигляді червоного еліпсу:

```
//Цей приклад припускає наявність Imports: System.Drawing
protected override void OnPaint(PaintEventArgs e) {
    Brush aBrush = new SolidBrush(Color.Red);
    Rectangle clientRectangle = new Rectangle(new Point(0,0),
this.Size);
    e.Graphics.FillEllipse(aBrush, clientRectangle);
}
```

При зміні розмірів елемента керування автоматично змінюється розмір **ClipRectangle**, але повторне промальовування цього елемента керування потрібно не завжди. Щоб при зміні розмірів елемента керування він кожен раз виконувався заново, викличте його з конструктора методом **Control.SetStyle** та встановіть **ResizeRedraw** у **true**, як показано в наступному прикладі:

```
SetStyle(ControlStyles.ResizeRedraw, true);
//Щоб у будь-який час перемалювати елемент керування вручну,
//достатньо викликати метод Refresh:
Refresh();
```

Як створити формований елемент керування:

- оголосіть клас, похідний від класу **Control**;
- додайте до методу **OnPaint** код, який відображає елемент керування;
- реалізуйте всі нестандартні можливості, необхідні вашому елементу керування.

2.2.3 Додавання елементів керування на панель **Toolbox**

Зручно, коли власні елементи керування завжди під рукою. Щоб зробити їх доступними при розробці будь-яких проектів та спростити роботу з ними, додайте їх на інструментальну панель **Toolbox**, звідки їх у будь-який час можна перетягнути на форму.

Це завдання дозволяє вирішити команда **Customize Toolbox**, доступна через контекстне меню панелі **Toolbox**. Натисніть правою кнопкою панель **Toolbox** та оберіть команду **Customize Toolbox**, щоб викликати діалогове вікно **Customize Toolbox**. У ньому оберіть завантажений елемент керування, який потрібно додати до панелі **Toolbox**.

Додавання елементу керування на інструментальну панель **Toolbox**:

- натисніть правою кнопкою миші панель **Toolbox** та оберіть команду **Customize Toolbox** – відкриється однойменне діалогове вікно;
- перейдіть до вкладки **.NET Framework Components** та натисніть на кнопку **Browse**, щоб викликати діалогове вікно **File**;
- відкрийте папку з DLL або EXE-файлом, у якому знаходиться потрібний елемент керування. Виділіть потрібний файл та натисніть кнопку **Open**;
- переконайтеся, що обраний елемент керування додано до списку діалогового вікна **Customize Toolbox** та натисніть **OK** – елемент керування з'явиться на панелі **Toolbox**.

Призначення значка елемента керування

Visual Studio .NET надає значок для відображення нестандартних елементів керування, доданих до панелі **Toolbox**, але можна призначити для цієї мети і власне растрове зображення. Значок для відображення на панелі **Toolbox** призначають за допомогою класу **ToolboxBitmapAttribute** – спеціалізованого атрибуту, що надає метадані елементу керування. Він дозволяє призначити елементу керування растрове зображення розміром **16x16** пікселів або задати значок за значенням **Type (type)**. У останньому випадку елементу керування присвоюється значок, прийнятий для об'єкту даного типу.

Атрибут **ToolboxBitmapAttribute** входить до оголошення класу елемента керування. У Visual C# цей атрибут розташовують у квадратні дужки до рядку оголошення класу елемента керування, але перед самим оголошенням. Приклад, як призначити елементу керування значок, вказавши файл з його зображенням:

```
[ToolboxBitmap @"C:\Pasta.bmp"]public class PastaMaker:Control
{
    //Реалізацію опущено
}
```

Як призначити елементу керування значок на основі його типу:

– додайте до оголошення класу елемента керування атрибут **ToolboxBitmapAttribute** з посиланням на тип даного елемента керування:

```
[ToolboxBitmap (typeof(Button))]public class myButton:Button
{
    //Реалізацію опущено
}
```

Налагодження елементів керування

У циклі розробки будь-якого компоненту рано чи пізно настає етап налагодження. Оскільки вони не є проектами, для налагодження їх необхідно помістити до спеціального проекту Windows Forms. Якщо ваш елемент керування є частиною проекту виконуваного компоненту, наприклад програми Windows Forms, для цього тестування слід додати до проекту нову форму, а якщо елемент керування входить до проекту бібліотеки класів або елементів керування, доведеться додати до вирішення додатковий проект.

Налагодження елементу керування в проєкті Windows Forms:

- виконайте компонування рішення, вибравши в меню Build команду Build Solution;
- при необхідності додайте до проєкту нову форму та зробіть її початковою формою проєкту;
- додайте елемент керування до форми через графічний інтерфейс або програмно. У першому випадку спочатку додають елемент керування до панелі **Toolbox**;
- запустіть програму, натиснувши F5, – тепер можна приступати до налагодження елементу керування, встановлення точок переривання, застосування покрокового виконання та інших засобів, що дозволяють знаходити помилки в коді;
- виправивши помилки, перекомпілюйте елемент керування, щоб внесені до нього зміни вступили в силу. Для цього оберіть у меню Build команду Rebuild Solution.

Налагодження елементу керування в проєкті бібліотеки класів або елементів керування:

- виконайте компонування рішення, вибравши в меню Build команду Build Solution;
- у меню File виберіть команду Add Project \ New Project, щоб додати до вирішення новий проєкт, – відкриється діалогове вікно Add New Project;
- у діалоговому вікні Add New Project виберіть WindowsApplication, дайте проєкту ім'я та натисніть **ОК**;
- у вікні Solution Explorer натисніть правою кнопкою миші вузол нового проєкту References та виберіть з контекстного меню команду Add Reference – відкриється діалогове вікно Add Reference;
- перейдіть до вкладки Projects. Якщо проєкт, що містить елемент керування, зазначено на цій вкладці, то виділіть його і натисніть (у іншому випадку натисніть кнопку Browse та знайдіть папку з DLL-файлом проєкту, в якому знаходиться відлагоджуваний елемент керування). Зробіть його виділеним та натисніть кнопку **ОК**;
- додайте елемент керування до форми через графічний інтерфейс або програмно. У першому випадку іноді спочатку доводиться додати елемент керування до панелі **Toolbox**, щоб спростити роботу з ним у період розробки.

Керування ліцензуванням елементів керування

У .NET Framework є вбудована підтримка керування ліцензуванням елементів керування. Ліцензування елементів керування дозволяє їх творцям захистити свою інтелектуальну власність, – надати доступ до своїх елементів керування тільки авторизованим розробникам. Модель ліцензування, прийнята за замовчуванням, вимагає щоб всі ліцензовані елементи керування були позначені атрибутом **LicenseProviderAttribute**, який вказує на об'єкт **LicenseProvider** – він необхідний для перевірки автентичності ліцензії. **LicenseProvider** – це абстрактний клас, що надає стандартний інтерфейс перевірки автентичності. У конструкторі елементу керування слід викликати метод **LicenseManager.Validate**, щоб отримати посилання на дійсну ліцензію. Якщо ліцензія до елементу керування відсутня, його завантаження не відбудеться. Метод **LicenseManager.Validate** викликає метод **GetLicense** заданого об'єкту **LicenseProvider**, який витягує ліцензію та перевіряє її за допомогою методу **IsKeyValid** об'єкту **LicenseProvider**. Якщо **IsKeyValid** повертає **true**, то ліцензія вважається дійсною, у протилежному випадку цей елемент керування не завантажуватиметься.

Застосована схема перевірки залежить від реалізації **LicenseProvider**. .NET – реалізація класу **LicenseProvider** має назву **LieFileLicenseProvider**. Якщо цей клас задано як **LicenseProvider**, він застосовується для перевірки ліцензії. Метод **GetLicense** цього класу шукає в каталозі з DLL-файлом елементу керування текстовий файл з ім'ям **FullName.LIC**, де **FullName** – повне ім'я елементу керування. Потім **IsKeyValid** перевіряє наявність у першому рядку текстового файлу підрядку «**myClassName is a licensed component**», де **myClassName** – повне ім'я компонента. Розробник легко замінить цей алгоритм перевірки власним, перевизначивши відповідні методи. У кожному ліцензованому елементі керування повинен бути метод **Dispose**, який слід викликати для звільнення ресурсів, пов'язаних з перевіркою ліцензії.

Приклад ліцензування елементу керування **Widget**, який використовує перевірку за допомогою **LieFileLicenseProvider**:

```
//Цей приклад припускає наявність оператора
//Imports System.ComponentModel
//Це атрибут, який вказує тип використаного LicenseProvider:
[LicenseProvider (typeof(LieFileLicenseProvider))]
public class Widget: System.Windows.Forms.Control {
    private License myLicense;
    public Widget() {
        //Перевірити ліцензію та отримати посилання до неї:
        myLicense = LicenseManager.Validate(typeof(Widget), this);
        //Інший код конструктора опущено
    }
    //Це метод Dispose, що звільняє ресурси,
    //які пов'язані з перевіркою ліцензії:
    protected override void Dispose(bool Disposing) {
        if (myLicense != null) {
            myLicense.Dispose();
            myLicense = null;
        }
    }
}
```

Реалізація керування ліцензуванням елементу керування:

- вкажіть клас **LicenseProvider**, додавши до оголошення класу атрибут **LicenseProviderAttribute**;
- створіть файл **License**, необхідний обраному класом **LicenseProvider**;
- у конструкторі елементу керування викличе метод **LicenseManager.Validate** для перевірки ліцензії;
- звільніть ресурси, зайняті перевіркою ліцензії, викликавши метод **Dispose** елементу керування.

2.3 Завдання до роботи

Ознайомитися з основними теоретичними відомостями, а також з друком та методами створення елементів керування при розробці текстового редактору на мові C#, використовуючи ці методичні вказівки, а також рекомендовану літературу.

2.4 Хід виконання роботи

Опрацювати та практично реалізувати кожен з прикладів, наведених у цьому розділі. Початковий код створювати самостійно та власноруч, застосовуючи в ньому власні оригінальні рішення.

Виконані напрацювання рекомендовано зберегти для подальшого використання у фінальній версії самостійної роботи щодо перегляду та роздрукування тексту і зображень, а також реалізації власних елементів керування.

2.5 Контрольні запитання

2.5.1 Які компоненти використовуються для реалізації друку?

2.5.2 Подія **PrintPage**, її властивості.

2.5.3 Як відбувається друк графіки?

2.5.4 Як відбувається друк тексту?

2.5.5 Елемент **PrintPreviewControl**, призначення та застосування.

2.5.6 Елемент **PrintPreviewDialog**, призначення та застосування.

2.5.7 Елемент **PageSettings**, призначення та конфігурування.

2.5.8 Елементи **PageSetupDialog** та **PrintDialog**, призначення та застосування.

2.5.9 Що таке **GDI+** та його призначення?

2.5.10 Яке призначення бібліотеки **System.Drawing**?

2.5.11 Детальний опис об'єкту **Graphics**, методи для малювання контурів та зафарбовування графічних елементів.

2.5.12 Детальний опис об'єктів **Color**, **Brush** та **Pen**, призначення, застосування і типи кистей.

2.5.13 Назвіть п'ять методів, щоб додати фігуру до об'єкту **GraphicsPath**.

2.5.14 Які відомі принципи створення елементів керування?

2.5.15 Як намалювати просту фігуру?

2.5.16 Як відбувається спадкування від класу **UserControl**?

2.5.17 Як відбувається спадкування від класу **Control**?

2.5.18 Як додавати створені компоненти до панелі **Toolbox**?

2.5.19 Як присвоїти елементу керування значок?

2.5.20 Як керувати ліцензуванням елементу керування?

3 РОБОТА З МЕНЮ ТА ОБРОБКА ВИНЯТКІВ У C#

3.1 Короткі теоретичні відомості

3.1.1 Меню та його пункти

Меню – це простий та звичний для користувачів інтерфейс, що відкриває доступ до команд та функцій верхнього рівня. Рационально сконструйоване меню впорядковує функціональність застосунку, що робить його легким для вивчення та використання. У іншому випадку, до невдало сконструйованому меню будуть звертатися тільки у випадку крайньої необхідності.

Контекстне меню

Контекстне меню зручно для доступу до набору команд, який залежить від поточного контексту програми. Контекстне меню створюють засобами компоненту **ContextMenu**. Для того щоб прив'язати його до певного об'єкту необхідно виділити об'єкт, при натисканні на який правою кнопкою буде з'являтися контекстне меню →, де у властивості **ContextMenuStrip** вибрати потрібний пункт контекстного меню.

Працюючи з меню можна перехопити змінну, яка містить назву вибраного пункту, її синтаксис:

```
MenuItem item = (MenuItem) sender;
//Змінній item присвоюється об'єкт sender, від якого прийшло
//повідомлення. Оскільки відомо, що повідомлення завжди
//посилаються від пунктів меню, виконується приведення до
//MenuItem.
string text = item.Text;
//Об'єкти класу MenuItem мають властивість Text.
```

Відмічені пункти меню

Якщо взяти до уваги стандартну програму ОС Windows – «Калькулятор» і заглянути до вкладки «Вигляд», то можна побачити відмічені пункти меню. Для того, щоб відмітити пункти меню потрібно вибрати у властивостях пункт **Checked** та поставити йому значення **true**, у пункті **CheckState** вибрати вигляд відмітки **Indeterminate** – вигляд крапки, **Checked** – вигляд прапорця, **Unchecked** – не показувати відмітку.

Для того, щоб в момент вибору пункту меню, потрібний пункт відмічався або переставав бути відміченим, оберіть потрібні пункти меню → змінити властивість **CheckOnClick** поставити в **true**.

Поділ елементів меню

При необхідності елементи меню відділяють один від одного горизонтальною лінією-роздільником. Роздільники дозволяють упорядкувати меню, яке складається з безлічі елементів, розбивши його на логічні групи.

Щоб додати до меню роздільник, введіть у полі меню дефіс замість імені, – під час виконання він перетвориться на роздільник.

Клавіші швидкого доступу

Натискаючи одночасно Alt і букву, призначену клавішею доступу, можна отримати швидкий доступ до пунктів меню. Скориставшись клавішею доступу, можна також вибрати команду з відкритого меню. Наприклад, у багатьох програмах Alt+F відкриває меню File. Визначити, яка буква призначена в якості клавіші доступу просто, – потрібна буква підкреслена в назвах елементів меню.

Дозволяється призначати однакові клавіші для доступу до елементів меню, якщо останні розташовано в різних групах. Наприклад, поєднання Alt+C застосовують для доступу до команд Close з групи File та Copy з групи Edit. Однак, не варто призначати однакові клавіші доступу до елементів меню з однієї групи. Якщо знехтувати цією рекомендацією, клавіша доступу буде по черзі активувати елементи меню для яких її призначено, але обрати який-небудь з них вдасться лише натисканням клавіші Enter.

Гарячі клавіші

Гарячу клавішу встановлюють значенням з перелічення **Shortcut** – це набір 150 клавіатурних сполучень, що рекомендовано для меню. Для звертання до пунктів меню за допомогою гарячих клавіш потрібно у властивостях знайти пункт **ShortKeys** та вибрати потрібне сполучення клавіш.

У Win32-програмуванні гарячі клавіші можна використовувати не тільки для швидкого виклику пунктів меню. Але у Windows Forms гарячі клавіші служать тільки для виклику пунктів меню. Гарячу

клавішу, яка не зв'язана з пунктом меню, можна визначити, зв'язавши спочатку пункт меню з цією клавішею, а потім додавши цей пункт у своє меню та встановивши властивість меню **Visible** до **false**. Пункт не буде відображатись, але швидка клавіша буде працювати.

Активация та деактивация команд меню

У кожного елементу меню є властивість **Enabled**. Встановивши її до **false**, деактивується елемент меню, забороняючи йому реагувати на дії користувача, у тому числі на клавіші доступу та швидкі клавіші. Деактивовані елементи меню виділені блідо-сірим кольором.

Вставка зображень для пунктів меню

Виділяється потрібний пункт меню та обирається властивість **Image**. У результаті з'явиться вікно обрання ресурсів (рис. 3.1):

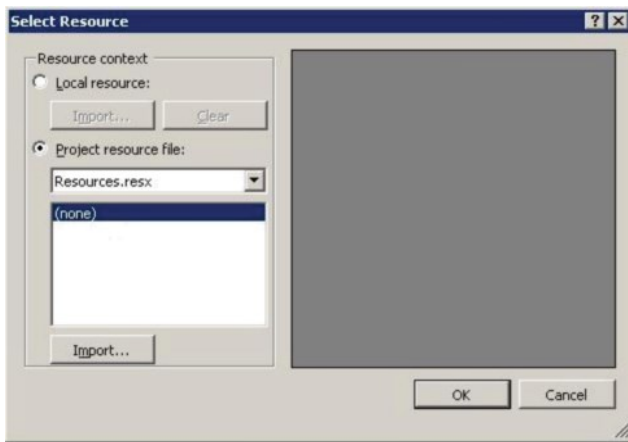


Рисунок 3.1 – Вікно для завантаження зображень пунктів меню

Надалі необхідно обрати один з варіантів:

Local resources – якщо вибрати цей пункт, то зображення не записуються у програму та будуть відкриватися з встановленого шляху. При перенесенні програми на інший комп'ютер необхідно буде разом з програмою скопіювати зображення, які було вибрано для цієї програми;

Project resource file – якщо вибрати цей пункт, то зображення зберігаються у програмі, і для перенесення програми на інший комп'ютер достатньо тільки скопіювати програму.

Створення меню програмно

Програмне створення меню, яке містить два пункти верхнього рівня меню «Файл» (з пунктами «Відкрити» та «Вихід») та Допомога (з пунктом «Про програму»), та при натисканні на пункт з'являється повідомлення:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication2 {
    public partial class Form1:Form {
        public Form1() {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e) {
            this.Text = "Просте меню";
            //Головне меню
            MenuStrip menu = new MenuStrip();
            this.Controls.Add(menu);
            //Розділ File
            ToolStripMenuItem itemFile = new ToolStripMenuItem();
            //Створення змінної, яка буде відповідати за пункт меню
            itemFile.Text = "&Файл";
            menu.Items.Add(itemFile);
            //Елемент верхнього рівня
            //Розділ Help
            ToolStripMenuItem itemHelp = new ToolStripMenuItem();
            itemHelp.Text = "&Допомога";
            menu.Items.Add(itemHelp);
            //Заповнення розділу File
            //Команда Open...
            ToolStripMenuItem itemOpen = new ToolStripMenuItem();
            itemOpen.Text = "&Відкрити...";
            itemOpen.ShortcutKeys = Keys.Control | Keys.O;
            itemOpen.ShowShortcutKeys = true;
            itemOpen.Click += OpenOnClick;
            itemFile.DropDownItems.Add(itemOpen);
            itemOpen.ShowShortcutKeys = false;
            //Роздільник
            ToolStripSeparator itemSep = new ToolStripSeparator();
            itemFile.DropDownItems.Add(itemSep);
```

```

//Команда Exit
ToolStripMenuItem itemExit = new ToolStripMenuItem();
itemExit.Text = "&Вихід";
itemExit.Click += ExitOnClick;
itemFile.DropDownItems.Add(itemExit);
//Заповнення розділу Help
//Команда About...
ToolStripMenuItem itemAbout = new ToolStripMenuItem();
itemAbout.Text = "&Про програму...";
itemAbout.ShortcutKeys = Keys.Control | Keys.A;
itemAbout.ShortcutKeyDisplayString = "Ctrl^A";
//Створення "гарячих" клавіш для цього меню
itemAbout.ShowShortcutKeys = true;
//Відображати в команді
itemAbout.Click += AboutOnClick;
itemHelp.DropDownItems.Add(itemAbout);
}
//Обробник подій
void OpenOnClick(object sender, EventArgs e) {
    MessageBox.Show("Заготівля опції \"Відкрити\"",
this.Text);
}
void ExitOnClick(object sender, EventArgs e) {
    this.Close();
}
void AboutOnClick(object sender, EventArgs e) {
    MessageBox.Show("Заготівля опції \"Про програму...\"",
this.Text);
}
}
}
}

```

3.1.2 Обробка та генерація винятків

Мова Visual C# підтримує структурну обробку винятків, що дозволяє створювати код, здатний коректно обробляти помилки та продовжити виконання програми. Крім того, можна створювати об'єкти винятків та генерувати винятки усередині компонентів для повідомлення основної програми про виникнення помилок.

Принципи обробки винятків

Виявивши помилку періоду виконання, застосунок генерує виняток (exception). Винятки становлять собою екземпляри спеціалізованих класів, нащадків класу **System.Exception**. Ці класи підтримують функції, спрощуючи діагностику помилок та керування їх обробкою, наприклад:

- властивість **Message** містить опис помилки в доступній для людини формі та інші важливі відомості про помилку;
- властивість **StackTrace** зберігає дані трасування стеку, забезпечує трасування для пошуку місця в коді, де виникла помилка.

Дозволяється створювати користувацькі винятки на основі класу **System.ApplicationException**. При виникненні помилки періоду виконання створюється екземпляр відповідного винятку та передається від методу, який згенерував помилку до гори по стеку викликів. У результаті, виняток отримує метод, з якого було викликано того, що згенерував помилку. Якщо в цьому методі є структура, здатна обробити цей виняток, вона обробляється, але у іншому випадку вона передається наступному методу в стеку викликів і т.д. Якщо структура обробила виняток та не знайшла його в стеку викликів, то викликається обробник винятків, якій задано за замовчуванням. Він відображає повідомлення із зазначенням типу винятку і будь-яких додаткових відомостей, наданих об'єктом винятку та закриває застосунок, не дозволяючи зберегти результати роботи чи виправити помилку. Застосування структурної обробки винятків дозволяє застосунку усунути наслідки непередбачених помилок або як мінімум, зберегти дані перед закриттям.

Створення обробників винятків

Обробники винятків реалізуються для окремих методів. У кожного методу повинен бути власний обробник винятків, орієнтований на особливості даного методу та пристосований для обробки винятків, які він зазвичай генерує. Обробку винятків неодмінно слід реалізувати в методах, які часто генерують виняток, наприклад використовують файловий доступ. Для обробників винятків прийнято синтаксис **try...catch...finally**.

try-блок інкапсулює код, що формує частину нормальних дій програми та в якому потенційно можуть відбутися серйозні помилкові ситуації.

catch-блок інкапсулює код, який оброблює помилкові ситуації, що відбуваються в коді блоку **try**.

finally-блок інкапсулює код, що очищує будь-які ресурси чи виконує інші дії, які необхідно виконати в кінці блоків **try** чи **catch**.

Основні етапи створення обробника винятків:

- помістіть код, який здатний генерувати винятки до блоку **try**;
- додайте один або декілька блоків **catch**, що обробляють різні типи винятків, які може генерувати код у блоці **try**;
- помістіть до блоку **finally** код, який необхідно виконати в будь-якому випадку. Цей код буде виконано незалежно від того, чи спрацювали винятки. Наприклад:

```
try {
    //Тут знаходиться звичайний код програми
}
catch {
    //Обробка помилок
}
finally {
    //Очищення
}
```

Оператор **catch** також дозволяє отримати посилання до об'єктів згенерованих винятків. Це посилання корисне, тому що в обробнику помилок через нього вдається отримати доступ до об'єкту винятків. Крім того, воно дозволяє перехоплювати певні типи помилок та створювати кілька блоків **catch**, призначених для обробки певних помилок.

До блоку **finally** поміщують код, який необхідно виконати незалежно від того, виникли винятки чи ні. Це може бути код, який зберігає данні, який звільняє ресурси або код, виконання якого є критичним для програмування. Приклад блоку **try...catch...finally** в програмі для ділення $1/x$, без операторів винятків виглядає так:

```
static void Main() {
    int x;
    int y = 1/x;
    Console.WriteLine(y);
}
```

Ця програма нормально функціонує при умові $x > 0$. Якщо застосувати оператори винятків, то програма отримає вигляд:

```
static void Main() {
    try {
        int x = int.Parse(Console.ReadLine());
        int y = 1/x;
        Console.WriteLine("y={0}", y);
        Console.WriteLine("блок try виконано вдало");
    }
```

```

catch {
    Console.WriteLine("виникла помилка");
}
Console.WriteLine("кінець програми");
}

```

Деякі класи обробки винятків наведено у таблиці 3.1.

Таблиця 3.1 – Класи обробки винятків

Назва	Опис
ArithmeticException	Помилка в арифметичних операціях чи перетвореннях
ArrayTypeMismatchException	Спроба збереження в масиві елементу несумісного типу
DivideByZeroException	Спроба ділення на нуль
FormatException	Спроба передати до методу аргумент невірнього діапазону
IndexOutOfRangeException	Індекс масиву виходить за границю діапазону
InvalidCastException	Помилка перетворення типу
OutOfMemoryException	Недостатньо пам'яті для нового об'єкту
OverflowException	Арифметичне переповнення
StackOverflowException	Переповнення стеку

Для того, щоб створити обробник винятків необхідно:

- додати до блоку **try** код, здатний генерувати помилки, які потрібно обробити;
- якщо винятки повинні оброблятися локально, створіть один або декілька блоків **catch** з відповідним кодом;
- код, який необхідно виконати незалежно від виникнення винятків помістіть до блоку **finally**.

Клас **Exception**

Основними властивостями класу є:

Message – рядок, що задає причину виникнення винятку. Значення цієї властивості встановлюється при виклику конструктора класу, коли створюється об'єкт, який задає виняток;

HelpLink – посилання (URL) до файлу, який містить докладну довідку про можливу причину виняткової ситуації та способи її усунення;

InnerException – посилання на внутрішній виняток. Коли обробник викидає новий виняток для передачі обробки на наступний рівень, то поточний виняток стає внутрішнім для новостворюваного винятку;

Source – ім'я програми, яка була причиною винятку;

StackTrace – низка викликів, методи, які зберігаються в стеку викликів у момент виникнення винятку;

TargetSite – метод, який викинув виняток.

З методів класу відзначається метод **GetBaseException**. При підйомі за низкою викликів він дозволяє отримати вихідний виняток – першопричину виникнення послідовності викиду винятків.

Клас має чотири конструктори, з яких три вже згадувалися. Один з них – конструктор без аргументів, другий – сприймає рядок, що стає властивістю **Message**, третій – має ще один аргумент, – виняток, який передано властивості **InnerException**.

У цьому випадку у створюваному винятку заповнюється властивість **InnerException**. Для стеження за властивостями винятків додано метод друку всіх властивостей, який викликається у всіх обробників винятків:

```
static public void PrintProperties(Exception e) {
    Console.WriteLine("Властивості винятку:");
    Console.WriteLine("TargetSite = {0}", e.TargetSite);
    Console.WriteLine("Source = {0}", e.Source);
    Console.WriteLine("Message = {0}", e.Message);
    if (e.InnerException == null)
        Console.WriteLine("InnerException = null");
    else
        Console.WriteLine("InnerException = {0}",
e.InnerException.Message);
    Console.WriteLine("StackTrace = {0}", e.StackTrace);
    Console.WriteLine("GetBaseException = {0}",
e.GetBaseException());
}
```

А тепер, два важливих правила:

а) обробка винятків повинна бути спрямована не стільки на повідомлення про виникнення помилки, скільки на коригування ситуації, яка виникла;

б) якщо виправити ситуацію не вдається, то програму необхідно перервати так, щоб не було отримано некоректних результатів, які не задовольняють специфікаціям програми.

Генерація винятків

Винятки можна розділити на два типи в залежності від ситуації:

а) при виникненні помилки періоду виконання, яку не вдається обробити повністю. Після часткової обробки такої помилки слід згенерувати виняток, щоб передати його по стеку для подальшої обробки;

б) при роботі компоненту виникає неприпустима ситуація, яку не вдається обробити локально. Про це необхідно повідомити батьківський компонент, згенерувавши стандартний виняток `.NET` або нестандартний виняток, специфічний для даного компоненту.

Повторна генерація винятків

Іноді структурі обробки винятків не вдається повністю обробити створений виняток. Наприклад, під час роботи методу, що виконує безліч складних обчислень, іноді несподівано виникає виняток **`NullReferenceException`**. У цьому випадку успішна обробка винятку можлива, як і вірогідний результат, навіть тоді, коли це не вдається. Початковий код повинен перевірити, чи в змозі він обробити, те що виникло. Якщо це не вдається в контексті блоку **`catch`**, то слід згенерувати його повторно за допомогою ключового слову **`throw`**, наприклад:

```
try {
    //Тут повинен бути код застосунку
}
catch (System.NullReferenceException e) {
    //Код перевірки винятку та поточний контекст, щоб визначити,
    //чи можна обробити виняток у даному блоці catch, передається
    //вище по стеку, якщо воно не може бути оброблено локально.
    //Локально наступний рядок генерує виняток повторно
    //і передає його далі по стеку.
    throw e;
}
```

При використанні універсального блоку **`catch`**, отриманий цим блоком виняток можна згенерувати повторно за допомогою ключового слову **`throw`**, вказавши його без аргументів.

У деяких випадках необхідно передати на наступний рівень програми додаткові відомості. Для цього замість повторної генерації винятку слід згенерувати інший виняток. У цьому випадку другий виняток грає роль оболонки для першого. Посилання до об'єкту

початкового винятку записується у властивість **InnerException** об'єкту нового винятку його конструктором. Цей прийом дозволяє передати з винятку системного типу додаткові дані, записавши їх до відповідних властивостей нового винятку та помістивши початковий виняток до властивості **InnerException**. Нижче демонструється створення та генерація нового винятку, який надає рядок з докладним описом початкового винятку, оболонкою якого він є:

```
//Це початок блоку catch:
catch (NullReferenceException e) {
    throw new NullReferenceException("якийсь текст", e);
}
```

Як повторно згенерувати виняток:

- щоб повторно згенерувати виняток, отриманий блоком **catch**, скористуйтеся ключовим словом **throw**;
- щоб згенерувати новий виняток, який надає додаткові відомості щодо початкового винятку, створіть об'єкт винятку, який є оболонкою для початкового винятку та згенеруйте новий виняток за допомогою ключового слову **throw**.

Нестандартний виняток

При розробці компонентів іноді також потрібно згенерувати виняток. Якщо під час роботи компоненту виникає неприпустима ситуація, слід згенерувати виняток для клієнтського застосунку. Для цих цілей завжди використовують нестандартний виняток.

Згідно назви, винятки призначені лише для не стандартизованих ситуацій. Не слід використовувати їх для взаємодії між клієнтом та компонентами – для цього призначені події. Виняток слід згенерувати, тільки коли продовження нормального виконання програми без зовнішнього втручання неможливо.

Нестандартний виняток можна створювати на основі класу **System.ApplicationException**. Цей клас інкапсулює всю необхідну функціональність, у тому числі властивості **Message**, **StackTrace** та **InnerException**, і може стати основою для нестандартних винятків, орієнтованих на потреби компонента.

Нижче наведено приклад нестандартного винятку, створеного на основі класу **ApplicationException**. Конструктор нового винятку вимагає фіктивний об'єкт **Widget**, оскільки новий виняток

надає доступ до цього об'єкту застосовуючи властивість, яка доступна лише для читання:

```
public class WidgetException: System.ApplicationException {
    //Ця змінна зберігає об'єкт Widget:
    Widget mWidget;
    public Widget ErrorWidget {
        get {
            return mWidget;
        }
    }
    //Цей конструктор приймає об'єкт Widget та рядок String,
    //ці параметри можна використовувати для опису контексту
    //програми на момент помилки. Дозволяється також створювати
    //перевантажені версії конструктору, що підтримують різні
    //набори параметрів.
    //Наступний рядок викликає конструктор з базового класу
    //та встановлює властивість Message, успадковане від класу
    public WidgetException(Widget W, string S):base(S) {
        //Встановити властивість Widget:
        mWidget = W;
    }
}
```

Оголосивши клас винятку, вдається згенерувати новий екземпляр цього класу при виникненні неприпустимої ситуації, наприклад, так:

```
Widget Alpha;
//Тут повинен бути код, що ушкоджує об'єкт Widget
throw new WidgetException(Alpha, "Widget Alpha is corrupt!");
```

Як згенерувати нестандартний виняток:

- оголосіть клас нестандартного винятку. Він повинен бути нащадком **System.ApplicationException** і може містити будь-які властивості, що надають додаткові дані клієнтам, які обробляють виняток. Ці властивості ініціалізуються у конструкторі класу нестандартного винятку;

- при виникненні нестандартної ситуації згенеруйте новий екземпляр класу нестандартного винятку за допомогою ключового слову **throw**, передавши конструктору відповідні параметри.

Генерація власних винятків

Щоб згенерувати власні винятки, необхідно скористатися оператором **throw**, вказавши параметри, які визначають вид винятку. Параметром має бути об'єкт, який породжено від стандартного класу **System.Exception**. Цей об'єкт використовується для передачі інформації щодо винятку обробника:

```
static void Main() {
    try {
        int x = int.Parse(Console.ReadLine());
        if (x < 0)
            throw new Exception();// 1
        Console.WriteLine("ok");
    }
    catch {
        Console.WriteLine("Введено некоректне значення");
    }
}
```

У рядку 1 за допомогою команди **new** було створено об'єкт винятку типу **Exception**. При необхідності можна згенерувати виняток будь-якого типу.

При генерації винятків можна визначити повідомлення, яке буде повертатися обробником винятків. Наприклад:

```
static void Main() {
    try {
        int x = int.Parse(Console.ReadLine());
        if (x < 0)
            throw new Exception("введено неприпустиме значення");// 1
        Console.WriteLine("ok");
    }
    catch (Exception error) {
        Console.WriteLine(error.Message);
    }
}
```

Корисні поради генерації винятків

Приклад 1. Один **try**-блок можна вкласти до іншого. Виняток, який згенеровано у внутрішньому **try**-блоці і не перехоплено **catch**-інструкцією, яка пов'язана з цим **try**-блоком, передається до зовнішнього **try**-блоку. Наприклад, у наступній програмі виняток типу **ArithmeticException** перехоплюється не внутрішнім **try**-блоком, а зовнішнім:

```
static void Main() {
    Console.WriteLine("a =");
    byte a = byte.Parse(Console.ReadLine());
    Console.WriteLine("b =");
    byte b = byte.Parse(Console.ReadLine());
    int f = 1;
    //Зовнішній try-блок
    try {
        for (byte i = a; i <= b; ++i) {
            //Внутрішній try-блок
            try {
```

```

        f = checked((int)(f*i));
        Console.WriteLine("y{0}) = {1:f6}", i, 100/(f-1));
    }
    catch (DivideByZeroException) {
        Console.WriteLine("y{0}) = Ділення на 0", i);
    }
}
}
}
catch (ArithmeticException)
    Console.WriteLine("ERROR");
}

```

Використання вкладених **try**-блоків обумовлено бажанням обробляти різні категорії помилок різними способами. Одні типи помилок носять катастрофічний характер і не підлягають виправленню. Інші – безпечні для подальшого функціонування програми і з ними можна справитися прямо на місці їх виникнення. Тому зовнішній **try**-блок можна використовувати для перехоплення самих серйозних помилок, дозволяючи внутрішнім **try**-блокам обробляти менш небезпечні.

Приклад 2. Виняток, перехоплений однією **catch**-інструкцією, можна згенерувати повторно, щоб забезпечити можливість його перехоплення іншою (зовнішньою) **catch**-інструкцією. Це дозволяє декільком обробникам отримати доступ до винятку:

```

static void genException() {
    Console.WriteLine("a =");
    double a = double.Parse(Console.ReadLine());
    Console.WriteLine("b =");
    double b = double.Parse(Console.ReadLine());
    int f = 1;
    //Зовнішній try-блок
    try {
        for (double i = a; i <= b; ++i) {
            //Внутрішній try-блок
            try {
                f = checked((int)(f*i));
                Console.WriteLine("y{0}) = {1:f6}", i, 100/(f-1));
            }
            catch (DivideByZeroException) {
                Console.WriteLine("y{0}) = Ділення на 0", i);
            }
        }
    }
}
}
catch (ArithmeticException) {
    Console.WriteLine("ERROR");
}

```

```

        //повторна генерація винятку:
        throw;
    }
}

static void Main() {
    try {
        genException();
    }
    catch {
        Console.WriteLine("фатальна помилка!!!");
    }
}

```

Потрібно пам'ятати, що при повторному генеруванні винятку він не буде повторно перехоплюватися той самою **catch**-інструкцією, а буде передано наступній (зовнішній) **catch**-інструкції.

Приклад 3. Як згадувалося вище, тип винятку повинен збігатися з типом, заданим у **catch**-інструкції. В іншому випадку цей виняток не буде перехоплено. Можна перехоплювати всі винятки, використовуючи **catch**-інструкцію без параметрів. Крім того, з **try**-блоком можна зв'язати не одну, а декілька **catch**-інструкцій. У цьому випадку всі **catch**-інструкції повинні перехоплювати винятки різного типу. Якщо немає впевненості, що передбачено всі ситуації, то на останню можна додати **catch**-інструкцію без параметрів.

Іноді виникає потреба в обов'язковому виконанні будь-яких дій, які повинні виконуватись щодо виходу з **try/catch**-блоку. Наприклад, генерується виняток і відбувається передчасне завершення виконання програмного фрагменту, але при цьому залишається відкритим файл. Для виходу з такої ситуації C# надає блок **finally**, який додається після всіх **catch**-блоків:

```

static void Main () {
    for (int i = 0; i < 5; i++) {
        try {
            Console.WriteLine("Введіть два числа");
            int a = int.Parse(Console.ReadLine());
            int b = int.Parse(Console.ReadLine());
            Console.WriteLine(a + "/" + b + "=" + a/b);
        }
        catch (FormatException) {
            Console.WriteLine("Потрібно ввести число!");
        }
        catch (DivideByZeroException) {
            Console.WriteLine("Ділити на нуль не можна!");
        }
    }
}

```

```

catch {
    Console.WriteLine("Якась помилка");
}
finally {
    Console.WriteLine("після try-блоку");
}
}
}

```

3.2 Завдання до роботи

Ознайомитися з основними теоретичними відомостями, а також з підходами та методами у розробці інженерного калькулятора на мові C#, використовуючи ці методичні вказівки, а також рекомендовану літературу.

3.3 Хід виконання роботи

Опрацювати та практично реалізувати кожен з прикладів, наведених у цьому розділі. Початковий код створювати самостійно та власноруч, застосовуючи в ньому власні оригінальні рішення. Виконані напрацювання рекомендовано зберегти для подальшого використання у фінальній версії самостійної роботи щодо реалізації головного і контекстного меню та обробки винятків.

3.4 Контрольні запитання

3.4.1 Яке основне призначення Головного та Контекстного меню?

3.4.2 Як присвоїти пунктам меню «гарячі» клавіші?

3.4.3 Як активувати та деактивувати пункти меню?

3.4.4 Як присвоїти пункту меню зображення?

3.4.5 Які існують операції для створення меню програмно?

3.4.6 Що таке виняток?

3.4.7 Як створити обробник винятку? Поясніть призначення кожного блоку винятку?

3.4.8 Який принцип обробки винятків?

3.4.9 Клас **Exception**. Основні властивості класу.

3.4.10 Навіщо потрібна повторна генерація винятків?

3.4.11 Як зробити повторну генерацію винятків?

3.4.12 Як згенерувати нестандартні винятки?

3.4.13 Навіщо використовувати вкладені **try**-блоки?

4 ЗАВДАННЯ ДЛЯ САМОСТІЙНИХ РОБІТ

4.1 Завдання до роботи

Використовуючи розділи 1–3 цих методичних вказівок для виконання самостійних робіт, студентам необхідно розробити завершені програмні рішення, які можна запустити на виконання як окремий застосунок.

У процесі створення застосунків потрібно розробити «дружній» користувацький інтерфейс (обов'язково із обробкою виключних ситуацій). Усі застосунки створюються тільки із використанням алгоритмічної мови C#.

4.2 Хід виконання роботи

4.2.1 Самостійна робота №1 – «Текстовий редактор»

Створити простий але зручний текстовий редактор з функціями форматування не тільки елементів тексту, але й інших OLE-об'єктів. У якості аналогів застосунку можна використовувати редактори «WordPad», «Microsoft Word» та інші подібні застосунки.

В залежності від насичення розроблюваного текстового редактору функціональністю, зручністю роботи з ним, відсутністю помилок, застосування у ньому прийомів, методів, засобів та технологій, буде сформовано адекватну оцінку за самостійну роботу.

Варіантом роботи є персональна оригінальність застосунку, за що буде підвищення оцінки, але й зменшення її, якщо буде підозра на плагіат.

4.2.2 Самостійна робота №2 – «Калькулятор»

Створити простий але зручний калькулятор з двома режимами роботи – «Інженерний» та «Програмістський», які містять відповідні функції (арифметичні, трансцендентні, тригонометричні, статистичні, логічні, тощо), а також з можливістю роботи у різних розмірностях (градуси, радіани, стерadianи) та системах обчислення (метрична, дюймова, Гаус, 2-, 8-, 10-, 16- та інших). У якості аналогів застосунку можна використовувати калькулятори, які вбудовано до ОС Windows, Linux, MacOS або інші подібні застосунки.

В залежності від насичення розроблюваного калькулятора функціональністю, зручністю роботи з ним, відсутністю помилок, застосування у ньому прийомів, методів, засобів та технологій, а також обов'язкової реалізації коректної обробки виняткових ситуацій, буде сформовано адекватну оцінку за самостійну роботу.

Варіантом роботи є персональна оригінальність застосунку, за що буде підвищення оцінки, але й зменшення її, якщо буде підозра на плагіат.

4.3 Зміст звіту

4.3.1 Сформульована мета роботи.

4.3.2 Постановка технічного завдання.

4.3.3 Алгоритм функціонування програми.

4.3.4 Лістинг готової програми.

4.3.5 Приклад роботи програми з обов'язковим наведенням копій екранів дисплею та опису представлених дій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Соммервил, И. Инженерия программного обеспечения [Текст]/ И. Соммервил. – М.: Издательский дом «Вильямс», 2002. – 623 с.
2. Pfleeger, S. Software Engineering. Theory and practice [Текст]/ S. Pfleeger. – New Jersey: Printice Hall, 1998. – 576 p.
3. Guide to the Software Engineering. Body of Knowledge (SWEBOOK). – New York: IEEE Publising House, 2004. – 129 p.
4. Орлов, С. Технологии разработки программного обеспечения. Учебник для Вузов [Текст]/ С. Орлов. – СПб.: Питер, 2002. – 463 с.
5. Лабор, В. Си Шарп. Создание приложений для Windows [Текст]/ В. Лабор. – Минск: Харвест, 2003. – 385 с.
6. Джонсон, Б. Основы Microsoft Visual Studio .Net 2003 [Текст]/ Б. Джонсон. – М.: Русская редакция, 2003. – 463 с.
7. Бишоп, Д. С# в кратком изложении [Текст]/ Д. Бишоп. – М.: БИНОМ Лаборатория знаний, 2005. – 467 с.