Міністерство освіти і науки України
Рівненський державний гуманітарний університет

Шахрайчук М.І., Шинкарчук Н.В., Шахрайчук А.М.

ОСНОВИ НАЛАШТУВАННЯ ЗАСТОСУВАНЬ У MICROSOFT VISUAL STUDIO .NET

Навчальний посібник

Рівне – 2018

УДК 004.4 (075.8)
Ш 32

Рекомендовано до друку Вченою радою Рівненського державного гуманітарного університету, протокол №2 від 22 лютого 2018 р.

Рецензенти:

Турбал Ю.В., доктор технічних наук, професор, Національний університет водного господарства та природокористування, м. Рівне;

Батишкіна Ю.В., кандидат технічних наук, доцент, Рівненський державний гуманітарний університет, м. Рівне.

Шахрайчук М.І., Шинкарчук Н.В., Шахрайчук А.М. Основи налаштування застосувань у Microsoft Visual Studio .NET: навчальний посібник / М.І. Шахрайчук, Н.В. Шинкарчук, А.М. Шахрайчук. – Рівне: РВВ РДГУ, 2018. – 192 с.

Розглянуті прийоми роботи в інтегрованому середовищі розробки Microsoft Visual Studio .NET, а також нові технології та інструменти середовища, призначені для налаштування ефективних сучасних застосувань. Як інструментальний засіб програмування використано мову C#, яка є однією із мов розробки програмного забезпечення у середовищі Microsoft Visual Studio .NET, та відлагоджувач .NET. Описана робота з рішеннями, проектами і редакторами коду. Посібник розрахований на студентів різних спеціальностей, які ґрунтовно вивчають програмування і в подальшому займатимуться створенням різноманітного програмного забезпечення. Матеріал супроводжується практичними прикладами і корисними порадами.

УДК 004.4 (075.8)

Ш 32

Навчальне видання

Шахрайчук Микола Іович,

Шинкарчук Назар Володимирович,

Шахрайчук Андрій Миколайович

ОСНОВИ НАЛАШТУВАННЯ ЗАСТОСУВАНЬ У MICROSOFT VISUAL STUDIO .NET

Навчальний посібник

Друкується в авторській редакції

Підписано до друку 22.02.2018 р.

Формат 60x84 1/16. Папір офсетний.

Гарнітура Times New Roman Cyr.

Ум. друк. арк. 8,7

Тираж 100 прим.

Замовлення № 529/2.

Відділ мережевого та інформаційного забезпечення

Рівненського державного гуманітарного університету.

33028, м. Рівне, вул. Ст. Бандери, 12.

ЗМІСТ

	Передмова	**7**
	Вступ	**9**
1	**Розділ 1. Архітектура .NET**	**13**
1.1	**Відносини між C# і .NET**	**18**
1.2	**Загальномовне виконавче середовище**	**19**
1.3	**Методи роботи трансляторів**	**20**
1.3.1	**Прямий метод**	**20**
1.3.2	**Метод розкручування**	**21**
1.3.3	**Віртуальна машина**	**21**
1.3.4	**Компіляція «на льоту»**	**22**
1.4	**Незалежність від платформи**	**23**
1.5	**Підвищення продуктивності**	**23**
1.6	**Мовна здатність до взаємодії**	**24**
1.6.1	**Visual C++**	**25**
1.6.2	**COM і COM+**	**26**
1.7	**Уважніший погляд на проміжну мову (IL)**	**26**
1.7.1	**Підтримка об'єктної орієнтації та інтерфейсів**	**27**
1.7.2	**Відмінність типів значень і типів посилань**	**29**
1.7.3	**Строга типізація даних**	**29**
1.7.3.1	**Строга типізація даних як ключ до міжмовної взаємодії**	**30**
1.7.3.2	**Загальна система типів**	**31**
1.7.3.3	**Загальна специфікація мови**	**31**
1.7.3.4	**Прибирання сміття**	**33**
1.7.3.5	**Безпека**	**35**
1.7.3.6	**Домени застосувань**	**36**
1.8	**Обробка помилок за допомогою виключень**	**39**
1.9	**Застосування атрибутів**	**40**
1.10	**Збірки**	**41**
1.10.1	**Приватні збірки**	**42**
1.10.2	**Розділювані збірки**	**43**
1.10.3	**Рефлексія**	**44**
1.10.4	**Паралельне програмування**	**44**
1.11	**Класи .NET Framework**	**45**
1.12	**Простори імен**	**46**
2	**Розділ 2. Інтегроване середовище розробки**	

	Microsoft Visual Studio .NET	**47**
2.1	**Огляд Visual Studio 2013 IDE**	**47**
2.1.1	**Посилання на початковій сторінці**	**47**
2.1.2	**Створення нового проекту**	**49**
2.2	**Види проектів**	**50**
2.3	**Основні частини візуального середовища розробки Visual Studio**	**52**
2.4	**Форми та елементи управління**	**53**
2.5	**Рядок меню і панель інструментів**	**56**
2.6	**Переміщення у Visual Studio IDE**	**60**
2.6.1	**Solution Explorer**	**60**
2.6.2	**Панель елементів**	**63**
2.6.3	**Вікно властивостей**	**64**
2.7	**Довідкова система**	**65**
2.7.1	**Контекстна підказка**	**66**
2.8	**Нескладне застосування з текстом і графікою**	**67**
2.9	**Підсумки**	**80**
3	**Розділ 3. Налаштування застосувань**	**81**
3.1	**Основні типи помилок**	**81**
3.1.1	**Синтаксичні помилки**	**81**
3.1.2	**Логічні помилки**	**82**
3.1.3	**Помилки періоду виконання**	**83**
3.2	**Відлагоджувач Visual Studio**	**86**
3.2.1	**Точки переривання**	**87**
3.2.1.1	**Налаштування точки переривання**	**88**
3.2.1.2	**Вікно Breakpoints**	**89**
3.2.1.3	**Налаштування точки переривання функції**	**91**
3.2.1.4	**Переривання на основі умов**	**93**
3.2.1.5	**Вікно File Breakpoint**	**94**
3.2.1.6	**Вікно Breakpoint Condition**	**94**
3.2.1.7	**Вікно Breakpoint Hit Count**	**97**
3.2.1.8	**Вікно Breakpoint Filter**	**98**
3.2.1.9	**Вікно When Breakpoint Is Hit**	**99**
3.3	**Покрокове проходження коду**	**101**
3.3.1	**Початок налаштування застосування**	**101**
3.3.2	**Проходження по коду**	**105**
3.3.3	**Продовження налаштування**	**106**
3.3.4	**Закінчення налаштування**	**107**

3.4	**Налагоджувальні вікна**	**108**
3.4.1	**Вікно Output**	**109**
3.4.2	**Вікно Locals**	**109**
3.4.3	**Вікно Autos**	**111**
3.5	**Функція налаштування IntelliTrace**	**112**
3.5.1	**Налаштування збору даних IntelliTrace для відлагодження у VS**	**112**
3.5.1.1	**Налаштування колекції подій**	**113**
3.5.1.2	**Налаштування колекції викликів функцій**	**115**
3.5.1.3	**Налаштування колекції модулів**	**117**
3.6	**Вікно Watch**	**117**
3.7	**Вікно QuickWatch**	**119**
3.8	**Вікно Command в режимі Immediate**	**120**
3.9	**Спливаючі підказки даних DataTips**	**122**
3.10	**Вікна візуалізації даних**	**122**
3.11	**Виключення**	**123**
3.12	**Класи Debug і Trace**	**124**
3.12.1	**Виведення трасування у вікно Output**	**125**
3.12.2	**Запис даних в набір Listeners**	**126**
3.12.3	**Перемикачі трасувань**	**134**
3.13	**Приклад налаштування за допомогою Visual Studio**	**139**
3.13.1	**Код, на прикладі якого демонструватимемо прийоми налаштування**	**139**
3.13.2	**Інструменти кодування, які спрощують розробку**	**143**
3.13.3	**Конфігурація налагоджувального режиму**	**146**
3.13.4	**Встановлення точок зупину**	**152**
3.13.4.1	**Створення точки зупину**	**152**
3.13.4.2	**Індивідуальне налаштування точки зупину**	**154**
3.13.4.3	**Управління точками зупину**	**156**
3.13.4.4	**Покрокове виконання коду**	**157**
3.13.5	**Дослідження стану застосування**	**160**
3.13.5.1	**Вікно Locals і Autos**	**161**
3.13.5.2	**Вікно Watch**	**162**
3.13.5.3	**Вікно Immediate**	**163**
3.13.5.4	**Вікно Call stack**	**163**
3.13.5.5	**Вікно Quick Watch**	**164**

3.13.5.6 Спостереження змінних з прив'язкою до джерела 165
3.13.5.7 Робота з IntelliTrace 167
3.13.6 Розв'язування проблем з допомогою відлагоджувача VS 169
3.13.6.1 Програма з помилками 169
3.13.6.2 Пошук помилок 175
3.13.6.3 Виправлення першої помилки 181
3.13.6.4 Налаштування і розв'язування проблем NullReferenceException 183
Підсумок 191
Використана література 192

ПЕРЕДМОВА

Помилки – це причини невдалих проектів із зірваними термінами. Помилки перетворюють життя програміста на кошмар, бо, якщо значне їх число зачаїться у програмному продукті, замовники продукту можуть припинити його використання, і це стане причиною втрати роботи. Помилки – це серйозний бізнес із серйозними наслідками; NASA втратила космічний зонд, спрямований на Марс, через помилки, допущені при напрацюванні вимог і проектуванні програмного забезпечення (ПЗ); на військових американського спецназу впала бомба, спрямована на іншу ціль. Причиною була програмна помилка, яка виникла при зміні джерела живлення у системі наведення. Через те, що комп'ютери управляють усе більш відповідальними системами, медичними пристроями і надзвичайно дорогою апаратурою, програмні помилки не можуть бути предметом гумору і не можуть розглядатися як щось само собою зрозуміле.

Цей навчальний посібник передусім допоможе зрозуміти, як налаштовувати програми з метою мінімізації числа помилок і передбачати причини їх виникнення. При правильному підході до процесу налаштування можна заощадити багато часу. Вважається, що налаштування треба розглядати не як окремий крок, а як складову частину загального циклу виробництва ПЗ. Розробники вважають, що над налаштуванням треба починати працювати на етапі формування вимог і продовжувати аж до стадії виробництва.

Налаштування в середовищах Microsoft .NET і Microsoft Windows *складні* і *затратні в часі*. Подолання *складності* вимагає досвіду, який можна набути шляхом самоосвіти, бо вивченню методів передбачення та усунення помилок практично не приділяється уваги у навчальних курсах присвячених опануванню програмування і створення ПЗ. Звернення до студентів: «Не пишіть програми з помилками!», не більше ніж бажання викладача та ідеал, до якого усі ми прагнемо. Насправді все злегка по іншому. Вивчення систематизованих перевірених методик налаштування не врятує від чергової помилки, але дотримання базових прийомів викладених у цьому посібнику допоможе вам скоротити число помилок, які вносяться в код, а ті з них, які все-таки туди прокралися, знайти швидше.

Суть часових *затрат* полягає в тому, що, незважаючи на величезне число чудових книг з окремих технологій .NET і Windows, ні в одній з них налаштуванню ПЗ або не приділяється уваги взагалі, або про нього (налаштування) згадується поверхнево. Для налаштування у рамках будь-якої технології треба знати значно більше, чим про окремі аспекти технології, яка описується в тій або іншій книзі. Одна справа знати, як вбудувати елемент управління на форму, зовсім інше – як повністю налаштувати цей елемент управління. Для його налаштування треба знати всі тонкощі середовища .NET і, наприклад, WPF (Windows Presentation Foundation) – графічна (презентаційна) підсистема в складі .NET Framework. Багато книг пояснюють реалізацію таких складних функцій, як з'єднання з віддаленою базою даних із застосуванням найсучасніших технологій, але коли у програмі не працює «db.Connect ("Foo")», – а рано чи пізно це обов'язково трапляється – треба самому розбиратися в усьому технологічному ланцюжку. Крім того, хоча є немало книг з управління проектами, в яких обговорюються питання налаштування, в них робиться наголос на управлінські та адміністративні проблеми, а не на завдання розробників. Ці книги можуть включати чудову інформацію про планування налаштування, але від цього мало користі, коли ви стикаєтеся з руйнацією бази даних або збоєм при поверненні з функції зворотного виклику.

Мета цього посібника – допомогти студентам, які навчаються «ремеслу» створення ПЗ різноманітного призначення, зрозуміти важливість опанування знаннями про інструментарій для здійснення налаштування та їх практичне використання у випадку, коли є розуміння того «що щось пішло не так», тобто логіка програми працює не за сценарієм автора. Витративши масу часу на пошук помилок автор приходить до розуміння того факту, що на «ура!» цю проблему не розв'яжеш, тобто треба шукати джерела, які розкажуть як же треба справлятися з такими проблемами. Можливо цей посібник буде поштовхом для розуміння студентами важливості використання інструментарію середовища .NET для з'ясування питання про те в якому місці коду щось пішло не так.

ВСТУП

Відлагодження (налаштування) – це процес виявлення та усунення причин помилок. У деяких проектах налаштування займає до 50% загального часу розробки. Багато програмістів вважають відлагодження найважчим аспектом програмування.

Але відлагодження не обов'язково має бути таким. Якщо використовувати інструменти, які надаються середовищем розробки, знати певні правила передбачення виникнення дефектів у коді програми, то число помилок стане значно меншим. Більшість дефектів будуть тривіальною неуважністю і помилки набору коду в редакторі, які можна легко знайти, читаючи початковий код або виконуючи його у відлагоджувачі. Що до інших, складніших помилок, то, вивчаючи поради «бувалих мисливців» за дефектами у програмному коді, використовуючи їх у своїй діяльності разом з інструментальними засобами, які надаються, наприклад, середовищем Microsoft Visual Studio .NET, ви зможете зробити їх відлагодження набагато легшим. Наведемо для прикладу два джерела в яких зосереджені поради, щодо методів прогнозування можливих місць появи дефектів та способи їх усунення: *Роббинс Дж. Отладка приложений для Microsoft .NET и Microsoft Windows* (2004 р.) та *Макконнелл С. Совершенный код* (2010 р.).

Дослідження показали, що досвідченим програмістам для знаходження тих же помилок, у порівнянні з недосвідченими, вимагалося приблизно в **20** разів менше часу. Більше того, деякі програмісти знаходять більше дефектів і виправляють їх грамотніше. Ось результати класичного дослідження, в якому професійних програмістів, які володіють мінімум **4**-річним досвідом, попросили налаштувати програму з **12** дефектами:

	Три найшвидших програмісти	Три найповільніших програмісти
Середній час відлагодження (у хвилинах)	5,0	14,1
Середнє число невиявлених дефектів	0,7	1,7
Середнє число нових	3,0	7,7

дефектів, внесених в код при виправленні наявних дефектів		
Джерело: «Some Psychological Evidence on How People Debug Computer Programs» (Gould, 1975)		

Три найкращих у відлагодженні програмісти змогли знайти дефекти приблизно в **3** рази швидше і внесли в код приблизно в **2,5** рази менше дефектів, чим три гірших. Найкращий програміст виявив всі дефекти і не вніс під час їх виправлення нових. Найгірший не знайшов **4** з **12** дефектів, а при виправленні **8** виявлених дефектів вніс в код **11** нових.

Після першого раунду налаштування в коді найшвидших програмістів все ще залишилися **3,7** дефектів, а в коді найповільніших – **9,4** дефектів. Жодна з груп не довела налаштування до кінця. Аналогічні великі відмінності в ефективності налаштування показали й інші дослідження (Gilb, 1977; Curtis, 1981).

Ці дані не лише повідомляють нам дещо про відлагодження, але і підтверджують Головний Закон Контролю Якості ПЗ: підвищення якості програми знижує витрати на її розробку. Кращі програмісти виявили більше дефектів за менший час і виправили їх коректніше. Немає потреби вибирати між якістю, вартістю і швидкістю – вони йдуть поруч.

Виходячи з того, що наявність дефектів у програмі небажана, робимо висновок, що мабуть автор програми не повністю розуміє програму. Це повинно непокоїти програміста, бо все-таки, якщо він написав програму, вона повинна робити те, що йому потрібно. Якщо програміст не до кінця розуміє свої вказівки компілятору, то через деякий час він просто почне пробувати все, що прийде в голову, поки щось не спрацює, т. т. почне програмувати методом спроб і помилок. А якщо так, дефекти неминучі. Можна подумати, що виправляти дефекти не потрібно – потрібно знати, як їх уникати.

Проте люди недосконалі, і навіть класні програмісти іноді допускають промахи; про початківців годі і говорити. У таких випадках помилки надають прекрасні можливості довідатися багато нового. Наприклад, ви можете зробити все, що описано нижче.

- **Вивчити програму, над якою працюєте.**
- **Вивчити власні помилки.**
- **Вивчити якість свого коду з погляду когось, кому доведеться читати його.**
- **Вивчити використовувані способи рішення проблем.**
- **Вивчити використовувані способи виправлення дефектів.**

Вище сказане дозволяє зробити висновок, що *налаштування забезпечує програмістам вкрай сприятливі можливості для самовдосконалення*. Саме відлагодження є перехрестям усіх доріг конструювання: легкості для читання, якості проекту і коду і т. д. Саме на етапі відлагодження окупається створення хорошого коду, особливо якщо його якість рідко примушує прибігати до відлагодження.

Основна мета цього посібника полягає в поясненні тонкощів функціонування інтегрованого середовища розробки Visual Studio .NET, у частині наявного інструментарію для виявлення дефектів програмного коду, їх аналізу та усунення.

Посібник складається з трьох розділів.

Розділ 1. Архітектура .NET.

Глави цього розділу дають уявлення про архітектуру Microsoft Visual Studio .NET, як основного середовища розробки ПЗ. Ті студенти, які знайомі з архітектурою .NET можуть просто переглянути ці глави.

Розділ 2. Інтегроване середовище розробки Microsoft Visual Studio .NET.

У цьому розділі описуються основні способи розробки у Visual Studio. Він дає розробникам базове розуміння багатьох функціональних можливостей їх основного інструменту. У главах розділу описуються більшість меню і вікон, які визначають кожен інструмент. Тут ми описуємо основні концепції проектів і рішень, а також детально досліджуємо провідники, редактори і візуальні конструктори.

Розділ 3. Налаштування застосувань.

Власне цей розділ присвячений опису інструментарію «заточеного» на виявлення ділянок коду у яких, можливо, зосереджені помилки. Детально розглядаються прийоми роботи з цим інструментарієм на конкретному модельному прикладі. Ретельно

аналізуються допущені помилки. Пропонуються ідеї удосконалення коду на етапі його створення, враховуючи виявлені дефекти, які привели до появи ділянок дефектного коду з метою недопущення помилок у майбутніх програмах з аналогічним кодом. Це власне і є той досвід, який набувається при відлагодженні програм, спрямований на покращення читабельності коду, якості проекту і коду і т. д.

РОЗДІЛ 1. АРХІТЕКТУРА .NET

Платформа **Microsoft .NET Framework** складається з **набору базових класів** і **CLR** (**Common Language Runtime**, *загальномовне середовище виконання*). Базові класи, які входять до складу **.NET Framework**, вражають своєю потужністю, універсальністю, зручністю використання і різноманітністю. У цьому розділі ми познайомимося з платформою **.NET** детальніше.

У **таблиці 1.1** відображені зміни версій платформи **.NET Framework** за останні роки, а у **таблиці 1.2** вибрані відомості про версії **.NET Framework**. Основою всього, звичайно, є *операційна система* (**ОС**). Найцікавіше, що Microsoft згадує тільки **Windows**-системи (Vista, 2000, XP і т. д.). У цьому списку немає інших платформ, хоча існує реалізація .NET для **Linux**, яка називається **Mono**. Просто ця реалізація випускається незалежним від Microsoft розробником, хоча і при істотній підтримці.

Таблиця 1.1. Версії **.NET Framework**

	Версія .NET Framework											
ОС	1.0	1.1	2.0	3.0	3.5	4.0	4.5	4.5.1	4.5.2	4.6	4.6.1	4.6.2
Windows 98	+											
Windows NT	+											
Windows Me	+											
Windows 2000	+	+	+									
Windows XP	+	+	+	+	+	+						
Windows Server 2003		+	+	+	+	+						
Windows Server 2008			+	+	+	+	+	+	+	+		
Windows Vista			+	+	+	+	+	+	+	+		
Windows 7			+	+	+	+	+	+	+	+	+	+
Windows Server 2008 R2			+	+	+	+	+	+	+	+	+	+
Windows			+	+	+	+	+	+	+	+	+	+

Server 2012												
Windows 8			+	+	+	+	+	+	+	+	+	+
Windows 8.1			+	+	+	+	+	+	+	+	+	+
Windows Server 2012 R2			+	+	+	+	+	+	+	+	+	+
Windows 10			+	+	+	+	+	+	+	+	+	+

Таблиця 1.2. Вибрані відомості про версії **.NET Framework**

Версії .NET Framework	**Опис**
.NET Framework 1.0	Перший реліз .NET Framework вийшов 5 січня 2002 року для Windows 98, NT 4.0, 2000 і XP.
.NET Framework 1.1	Наступний реліз .NET Framework вийшов 1 квітня 2003 року. Це була перша версія, яка автоматично встановлюється разом з операційною системою (Windows Server 2003). Для старіших операційних систем .NET Framework 1.1 була доступна у вигляді окремого інсталяційного пакета.
.NET Framework 2.0	Версія 2.0 була випущена одночасно з Visual Studio 2005, SQL Server 2005 і BizTalk 2006. З виходом версії 2.0 була додана підтримка налаштовуваних класів, анонімних методів, повна підтримка 64-бітових платформ x64 і IA-64.
.NET Framework 3.0	Спочатку .NET Framework 3.0 носила ім'я WinFX, що відбивало її суть: розширення .NET Framework 2.0 зі збереженням усіх бібліотек і додаванням чотирьох нових компонентів: ➢ Windows Presentation Foundation (WPF) – презентаційна графічна підсистема, яка використовує XAML; ➢ Windows Communication Foundation (WCF) – уніфікована програмна модель міжплатформної взаємодії; ➢ Windows Workflow Foundation (WF) –

	технологія визначення, виконання та управління робочими процесами; ➢ Windows CardSpace – технологія уніфікованої ідентифікації.
.NET Framework 3.5	Як і версія 3.0, .NET 3.5 використовує CLR версії 2.0. Нововведення у порівнянні з .NET Framework 3.0 включають: ➢ C# 3.0 і VB.NET 9.0; ➢ Додана мова LINQ і провайдери LINQ to Objects, LINQ to XML і LINQ to SQL; ➢ Додано ASP.NET AJAX; ➢ Розширена функціональність WF і WCF; ➢ Доданий простір імен System.CodeDom.
.NET Framework 4.0	Microsoft анонсувала .NET 4.0 29 вересня 2008 року. Перша бета-версія з'явилася 20 травня 2009 року, разом з бета-версією й Visual Studio 2010. Нововведення включають: ➢ Parallel Extensions – PLINQ (Parallel LINQ) і бібліотеку паралельних завдань (Task Parallel Library), призначені для спрощення програмування для багатопроцесорних і розподілених систем; ➢ Нововведення у Visual Basic і C#; ➢ Технологію Managed Extensibility Framework (MEF); ➢ Повну підтримку IronPython, IronRuby і F#; ➢ Підтримку підмножин .NET Framework і ASP.NET у варіанті Server Core; ➢ Підтримку Code Contracts; ➢ Засоби моделювання Oslo і мова програмування M, призначена для створення предметно-орієнтованих мов і моделей; Остаточна версія .NET Framework 4.0 була випущена 12 квітня 2010 року разом з кінцевою версією Visual Studio 2010.
.NET Framework 4.5	При встановленні замінює .NET Framework 4.0. Несумісна з Windows XP і більш ранніми версіями Windows. Нововведення включають:

	➢ Можливість зменшення числа перезапусків системи шляхом виявлення і закриття застосувань платформи .NET Framework версії 4 під час розгортання; ➢ Підтримка масивів, розмір яких перевищує 2 гігабайти на 64-розрядних платформах; ➢ Поліпшена продуктивність завдяки фоновому прибиранню сміття для серверів. При використанні серверного збирання сміття в .NET Framework 4.5 фонове прибирання сміття включається автоматично; ➢ Фонова компіляція на вимогу (JIT), яка опціонально доступна на багатоядерних процесорах для підвищення продуктивності застосування; ➢ Можливість обмежити час роботи обробника регулярних виразів при спробі обчислити регулярний вираз до завершення часу очікування; ➢ Можливість визначити культуру за замовчуванням для домена застосування; ➢ Підтримка кодування Юнікод (UTF–16) в консолі; ➢ Підтримка управління версіями культурних даних сортування і порівняння рядків; ➢ Поліпшена продуктивність при отриманні ресурсів; ➢ Поліпшення стискування ZIP;. ➢ Можливість налаштовувати контекст відображення для перевизначення поведінки відображення за замовчуванням; ➢ Підтримка версії 2008 стандарту інтернаціоналізованих доменних імен в застосуваннях, коли клас використовується у Windows 8; ➢ Делегування порівняння рядків ОС, яка реалізує Юнікод 6.0, якщо платформа .NET Framework використовується у Windows 8.

	При роботі на інших платформах .NET Framework включає власні відомості про порівняння рядків, які реалізують Юнікод 5.xx. ➢ Можливість обчислення хеш-коду для рядків на основі домена для кожного застосування; ➢ Підтримка JSON
.NET Framework 4.5.1	.NET Framework 4.5.1 випущений 17 жовтня 2013 року разом з Visual Studio 2013. Ця версія вимагає Windows Vista SP2 або свіжішу версію, і постачається разом з Windows 8.1 і Windows Server 2012 R2.
.NET Framework 4.5.2	.NET Framework 4.5.2 є оновленням .NET Framework 4.5.1, .NET Framework 4.5, і .NET Framework 4. Встановлюється, при потребі, разом з .NET Framework 3.5 Service Pack 1.
.NET Framework 4.6	.NET Framework 4.6 є оновленням .NET Framework 4.5.2, .NET Framework 4.5.1, .NET Framework 4.5, і .NET Framework 4. Встановлюється, при потребі, разом з .NET Framework 3.5 Service Pack 1. Є частиною редакції Microsoft Visual Studio 2015. .NET Framework 4.6 підтримує новий JIT-компілятор для 64-розрядних систем (RyuJIT); WPF і WinForms оновлені для підтримки екранів з високим DPI; у WCF була додана підтримка TLS 1.1 і TLS 1.2. Криптографічний API в .NET Framework 4.6 використовує останню версію API від Microsoft CryptoAPI, завдяки цьому став доступний набір алгоритмів шифрування «Suite B» – AES, SHA – 2, Elliptic curve Diffie – Hellman, ECDSA.
.NET Framework 4.6.1	.NET Framework 4.6.1 є оновленням .NET Framework 4.6, .NET Framework 4.5.2, .NET Framework 4.5.1, .NET Framework 4.5, і .NET Framework 4. Встановлюється, при потребі, разом з .NET Framework 3.5 Service Pack 1. Є частиною редакції Microsoft Visual Studio 2015

	Update 1.
.NET Framework 4.6.2	.NET Framework 4.6.2 був анонсований 30 березня 2016 року і випущений 2 серпня 2016 року. Є оновленням .NET Framework версій 4.6.1, 4.6, 4.5.2, 4.5.1, 4.5 і 4. Для встановлення потрібний Windows 7 SP1 або вище. Встановлюється, при потребі, разом з .NET Framework 3.5 Service Pack 1.

Платформа .NET складається з багатьох різних технологій, серверів і постійно розширюється. Варто тільки подивитися на складові .NET Framework, які постійно з'являються, і можна зрозуміти, що платформа розширюватиметься в майбутньому.

Важливо зауважити, що мову програмування C# треба розглядати паралельно із середовищем .NET. Компілятор C# спеціально націлений на .NET, а це означає, що *увесь код, написаний на C#, завжди виконуватиметься тільки у середовищі .NET*. Звідси випливає два важливі висновки щодо мови C#.

1. Архітектура і методологія C# віддзеркалює методологію .NET, яка лежить в його основі.
2. У багатьох випадках специфічні засоби мови C# насправді залежать від засобів .NET, тобто від базових класів .NET.

Через існування цієї залежності, перш ніж приступити до програмуванню на мові C#, важливо отримати деяке уявлення про архітектуру і методологію .NET. Це і буде метою цього розділу.

1.1. Відносини між C# і .NET

C# – це відносно нова мова програмування, яка характеризується двома наступними перевагами:

1. **C#** спроектований і розроблений спеціально для використання з **Microsoft .NET Framework** (розвиненою платформою розробки, розгортання і виконання розподілених застосувань).

2. **C#** – мова, яка базується *на сучасній об'єктно-орієнтованій методології проектування*, при розробці якої фахівці з Microsoft спиралися на досвід створення подібних мов, побудованих відповідно до запропонованих близько 20 років тому об'єктно-орієнтованих принципів.

Треба підкреслити ту важливу обставину, що C# – це повноцінна мова програмування. Хоча вона і призначена для генерації коду, який виконується в середовищі .NET, сама по собі вона не є частиною .NET. Існує *ряд засобів*, які *підтримуються* .NET, але *не підтримуються* C#, та існують також засоби, підтримувані C# і не підтримувані .NET (наприклад, деякі випадки перевантаження операцій). Проте через те, що мова C# призначена для використання на платформі .NET, розробникам, важливо мати уявлення про .NET Framework, якщо вони хочуть ефективно розробляти застосування на C#. Тому ми витратимо деякий час на те, щоб заглянути «за куліси» .NET.

1.2. Загальномовне виконавче середовище

Центральною частиною каркасу .NET є його *загальномовне виконавче середовище*, відоме як **Common Language Runtime** (CLR) або **.NET runtime**. Код, який виконується під управлінням **CLR**, часто називають *керованим кодом*.

Проте перш ніж код зможе виконуватися **CLR**, будь-який початковий текст (на C# або іншій мові) має бути скомпільований. Компіляція в .NET складається з двох кроків:

1. Компіляція початкового коду в Microsoft **Intermediate Language** (IL).
2. Компіляція **IL** в специфічний для платформи код за допомогою **CLR**.

Цей двокроковий процес компіляції дуже важливий, тому що наявність Microsoft **Intermediate Language** (IL) є ключем до багатьох переваг .NET.

Microsoft **Intermediate Language** (*проміжна мова* Microsoft) розділяє з *байт-кодом* **Java-***ідею низькорівневої мови* з простим синтаксисом (який базується на числових, а не текстових кодах),

який може бути дуже швидко трансльований в рідний машинний код. Наявність цього коду з чітко визначеним універсальним синтаксисом дає ряд істотних переваг.

1.3. Методи роботи трансляторів

Для кращого розуміння компіляції початкового коду в Microsoft **Intermediate Language** (IL) розглянемо методи роботи трансляторів. Серед методів варто розглянути наступні:

- Прямий.
- Розкручування.
- Крос-транслятор.
- Віртуальна машина.
- Компіляція «на льоту».

1.3.1. Прямий метод

Прямий (інтерпретація)

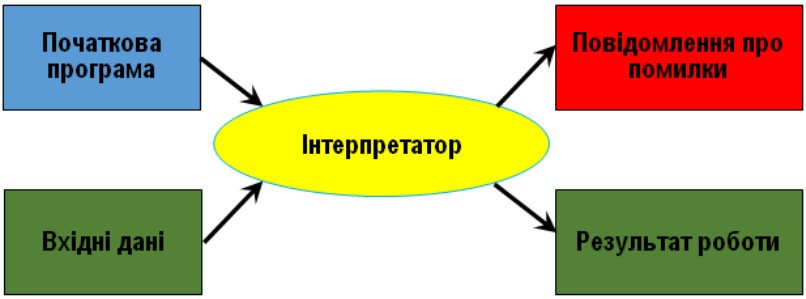


Рис. 1.1. Метод прямої інтерпретації

Прямий (компіляція)

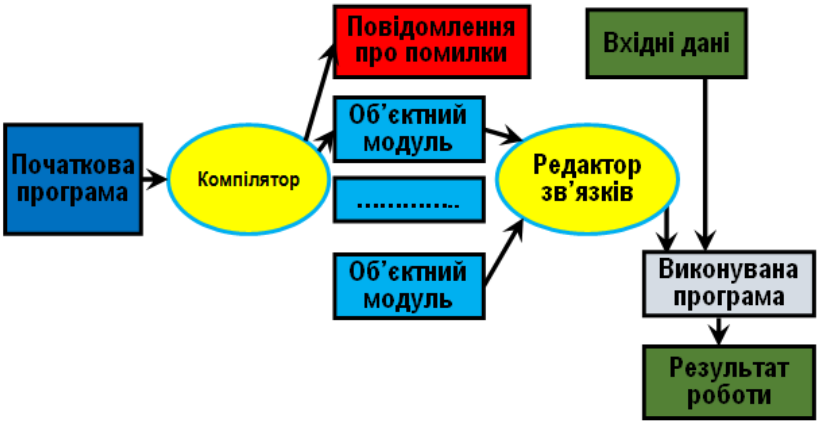


Рис. 1.2. Метод прямої компіляції

1.3.2. Метод розкручування

- **Розкручування** – *послідовне використання декількох компіляторів* для отримання цільової програми на потрібній мові.
- **Крос-транслятор** – компіляція коду *для однієї платформи* в цільову програму *для іншої платформи*.

1.3.3. Віртуальна машина

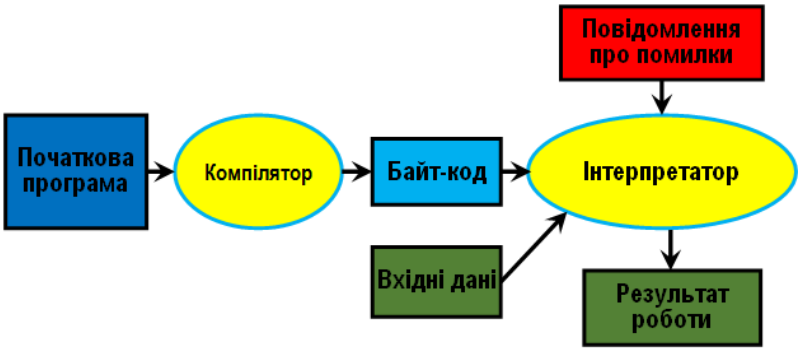


Рис. 1.3. Метод віртуальної машини

1.3.4. Компіляція «на льоту»

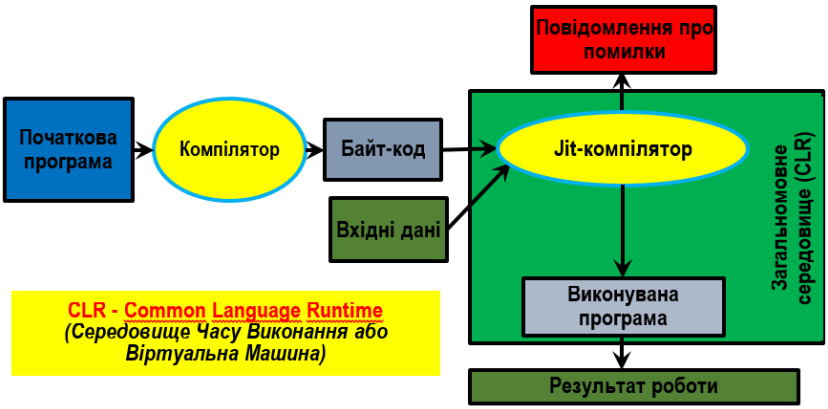


Рис. 1.4. Метод компіляції «на льоту»

Компанія Microsoft розробила компілятори для наступних мов програмування, які використовуються на цій платформі: **C++/CLI, C#, Visual Basic, F#, Iron Python, Iron Ruby** та *асемблер* **Intermediate Language** (IL). Окрім Microsoft, ще декілька компаній та університетів створили компілятори, призначені для середовища виконання **CLR**. Відомі компілятори для **Ada, APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML, Mondrian, Oberon, Pascal, Perl, Php, Prolog, RPG, Scheme, Smalltalk** і **Tcl/Tk**.

Процес компіляції файлів з початковим кодом зображено на **рис. 1.5**. Як видно з рисунка, *початковий код* програми може бути написаний *на будь-якій мові*, яка підтримує середовище виконання **CLR**. Незалежно від типу використовуваного компілятора результатом компіляції буде ***керований модуль*** (*managed module*) – стандартний ***переносний виконуваний*** (*portable executable*, **PE**) файл **32**-*розрядної* (**PE32**) або **64**-*розрядної* Windows (**PE32+**), який вимагає для свого виконання **CLR**. До речі, керовані збірки завжди використовують переваги функції безпеки «***запобігання виконанню даних»*** (**DEP, Data Execution Prevention**) і ***технологію*** **ASLR** (**Address Space Layout Optimization**), застосування цих технологій підвищує інформаційну безпеку усієї системи.

Компілятори машинного коду *генерують код, для конкретної процесорної архітектури* (**x86, x64** або **ARM)**. На відміну від цього, *всі* **CLR**-*сумісні компілятори генерують* **IL-код**. **IL-код** іноді називають ***керованим*** (*managed code*), тому що **CLR** управляє його виконанням.

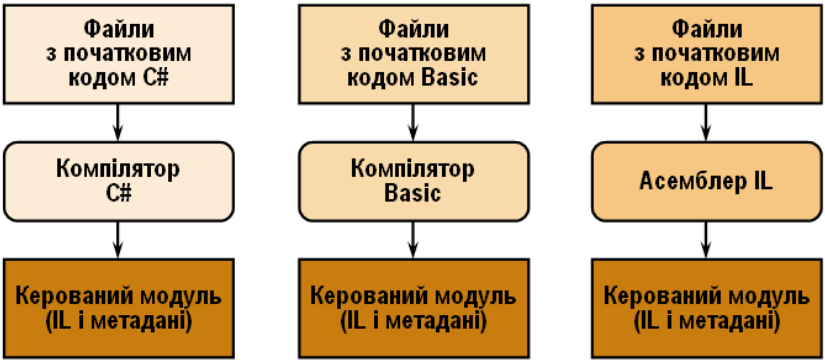


Рис. 1.5. Процес компіляції файлів у середовищі **CLR**

1.4. Незалежність від платформи

Насамперед, це означає, що файл, який містить інструкції байт-коду, може бути розміщений на будь-якій платформі; під час виконання може бути легко проведена фінальна стадія компіляції, що дозволить виконати код на конкретній платформі. Іншими словами, компілюючи в **IL**, ви отримуєте платформну незалежність .NET – багато в чому так само, як компіляція у байт-код **Java** забезпечує незалежність від платформи програмам на **Java**.

1.5. Підвищення продуктивності

Хоча мова **IL** вище порівнювалася з **Java**, все ж **IL** насправді гнучкіша, ніж байт-код **Java**. Код **IL** завжди *компілюється оперативно* (**Just-In-Time**, **JIT**-*компіляція*), у той час як байт-код **Java** *часто інтерпретується* (див. **рис. 1.3**). Одним з недоліків **Java** було те, що під час виконання програм процес трансляції байт-коду **Java** в рідний машинний код приводив до зниження продуктивності (за винятком самих останніх версій, де **Java** *компілюється оперативно* (**JIT**) на деяких платформах).

Замість компіляції усього застосування за *один прохід* (що може привести до затримок при запуску), **JIT**-компілятор просто *компілює кожну порцію коду при її виклику* (т. т. *оперативно*). Якщо проміжний код одного разу скомпільований, то результуючий машинний виконуваний код зберігається до моменту завершення роботи застосування, тому його перекомпіляцію при повторних зверненнях до нього не потрібно здійснювати. У Microsoft аргументують, що такий процес ефективніший, ніж компіляція усього застосування при запуску, бо існує висока вірогідність того, що великі фрагменти коду застосування насправді не виконуватимуться при кожному запуску. При використанні **JIT**-компілятора такий код ніколи не буде скомпільований.

Це пояснює, чому можна розраховувати на те, що виконання керованого коду **IL** буде майже настільки ж швидким, як і виконання рідного машинного коду. Проте це не пояснює того, чому Microsoft чекає підвищення продуктивності. Причина полягає в тому, що через те, що фінальна стадія компіляції відбувається під час виконання, **JIT**-компілятор на цей момент вже знає, на якому типі процесора буде запущена програма. А це означає, що він може оптимізувати фінальний виконуваний код, використовуючи інструкції конкретного машинного коду, призначені для конкретного процесора.

Традиційні компілятори оптимізують код, але вони можуть здійснювати лише оптимізацію, незалежну від конкретного процесора, на якому код виконуватиметься. Це відбувається тому, що традиційні компілятори генерують виконуваний код до того, як він постачається користувачам. А тому компілятору не відомо, на якому типі процесора вони працюватимуть, за винятком найзагальніших характеристик на кшталт того, що це буде **x86**-сумісний процесор або ж процесор **Alpha**.

1.6. Мовна здатність до взаємодії

Використання **IL** *не лише забезпечує незалежність від платформи*; вона також *забезпечує здатність до взаємодії*. Тільки уявіть, що ви можете скомпілювати програму на одній мові в код **IL**, і цей скомпільований код зможе вільно взаємодіяти з **IL**-кодом, скомпільованим з іншої мови.

Можливо, той факт, що з .NET *сумісні й інші мови* окрім C#, викликає подив, тому ми коротко обговоримо, як деякі мови загального призначення стикуються з .NET

1.6.1. Visual C++

Мова **Visual C++ 6** (ті хто працював з цією мовою розуміють про що йдеться) вже мала багато специфічних для Microsoft розширень Windows. З появою **Visual C++ .NET** були додані нові розширення, які забезпечують підтримку середовища .NET. Це означає, що ***існуючий*** *початковий код* C++ буде як і раніше компілюватися в рідний машинний код без всяких модифікацій. Проте це також означає, що він працюватиме незалежно від виконавчого середовища .NET. Якщо треба, щоб програми C++ запускалися всередині середовища .NET, можна помістити в початок початкового коду наступний рядок:

```
#using <mscorlib.dll>
```

Можна також передати компілятору прапорець **/clr**, який припускає, що повинна здійснюватися компіляція в керований код, тому замість рідного машинного коду генерується код **IL**. Цікаво, що при компіляції C++ в керований код компілятор може згенерувати **IL**, який включає виконуваний машинний код. Це означає, що в коді на C++ *можна змішувати* ***керовані*** (*managed*) типи *з* ***некерованими*** (*unmanaged*). Тобто, керований код C++

```
class MyClass
{
```

оголошує *простий клас* C++, тоді як код

```
ref class MyClass
{
```

дасть *керований клас*, начебто він був написаний на C# або Visual Basic. Перевага застосування керованого коду C++ перед C# пов’язана з тим, що можна викликати некеровані класи C++ з

керованого коду C++ без потреби звертатися до засобів взаємодії з **COM**.

Компілятор видає помилку при спробі використання засобів, які не підтримуються .NET або керованими типами (наприклад, шаблонів або множинного спадкоємство класів). Крім того, при роботі з керованими класами ви зіткнетеся з потребою застосування нестандартних засобів C++.

Через свободу, яку надає C++ відносно низькорівневих маніпуляцій з покажчиками і тому подібного, компілятор C++ не може генерувати код, який успішно пройде тести **CLR** *на безпеку типів*. Якщо важливо, щоб код розпізнавався **CLR** *як безпечний щодо типів*, при написанні початкового коду краще віддати перевагу якійсь іншій мові (такій як C#, або Visual Basic).

1.6.2. COM і COM+

Формально **COM** і **COM+** не є технологіями, націленими на .NET, бо компоненти, які базуються на них, не можуть компілюватися в **IL** (хоча до певної міри це і можна зробити, застосовуючи керований C++, якщо початковий компонент **COM+** був написаний на C++). Проте **COM+** залишається важливим інструментом, тому що *його засоби не дублюють* .NET. До того ж компоненти **COM** як і раніше працюватимуть, і .NET включає засоби взаємодії з **COM**, які дозволяють керованому коду викликати компоненти **COM** і навпаки. Проте, швидше за все, виявиться, що у більшості випадків зручніше кодувати нові компоненти у вигляді компонентів .NET, *щоб скористатися перевагами базових класів* .NET, а також іншими вигодами від запуску керованого коду.

1.7. Уважніший погляд на проміжну мову (IL)

Як ви зрозуміли з попереднього викладу, проміжна мова Microsoft очевидно грає фундаментальну роль в середовищі .NET. Тепер є резон уважніше розглянути основні характеристики **IL**, бо логічно, що будь-яка мова, призначена для .NET, також повинна підтримувати ці основні характеристики IL.

Найважливіші властивості **IL** можуть бути сформульовані таким чином:

- об’єктна орієнтація і застосування інтерфейсів;

- строга відмінність між типами значень і типами посилань;
- строга типізація даних;
- обробка помилок шляхом використання виключень;
- застосування атрибутів.

Усі перераховані властивості розглядаються в наступних розділах.

1.7.1. Підтримка об'єктної орієнтації та інтерфейсів

Незалежність .NET від мови має деякі практичні обмеження. **IL** неминуче повинна втілювати деяку визначену методологію програмування, а це означає, що початкова мова також має бути сумісна з цією методологією. Принцип, яким керувалися в Microsoft при створенні **IL**: *класичне об'єктно-орієнтоване програмування з реалізацією одиночного спадкоємства класів*.

На додаток до класичної об'єктної орієнтації в мові **IL** також *введено поняття інтерфейсів*, які уперше були реалізовані під Windows з появою **COM**. Інтерфейси, побудовані з використанням **.NET** – *це не те ж саме*, що інтерфейси **COM**; їм не потрібна підтримка з боку інфраструктури **COM** (так, наприклад, вони не спадкують від **IUnknown** і не мають асоційованих глобальних ідентифікаторів **GUID**). Проте *вони розділяють* з інтерфейсами **COM** ідею надання контракту, і класи, які реалізовують заданий інтерфейс, повинні пропонувати реалізацію методів і властивостей, специфікованих цим інтерфейсом.

Ви вже знаєте, що робота з .NET припускає компіляцію в **IL**, а це, у свою чергу, означає потребу у дотриманні об'єктно-орієнтованої методології. Проте тільки цього недостатньо, щоб *забезпечити міжмовну взаємодію*. Врешті-решт, **C++** і **Java** також використовують об'єктно-орієнтовану парадигму, але *їх не можна назвати здатними до взаємодії*. Треба детальніше розглянути концепцію мовної взаємодії.

Так що ж у загальному випадку мається на увазі під мовною взаємодією?

Врешті-решт, вже технологія **COM** дозволяла писати компоненти різними мовами, забезпечуючи їх спільну роботу – в тому сенсі, що вони могли викликати методи один іншого. Чого ж бракувало? Технологія **COM**, *будучи бінарним стандартом*, не

дозволяла компонентам створювати екземпляри інших компонентів і викликати їх методи і властивості, не піклуючись про мову, на якій компоненти були написані. Щоб забезпечити таку можливість, кожен об'єкт повинен був створюватися через виконавче середовище **COM**, і бути доступним через інтерфейс. Залежно від моделей потоків взаємодіючих компонентів, вони могли приводити до істотних втрат продуктивності, які пов'язані з передачею даних між апартаментами або компонентами, які виконуються, в різних потоках. У крайньому випадку, коли компоненти розміщувалися у виконуваних файлах, а не в **DLL**, для їх виконання доводилося запускати окремі процеси. Постійно підкреслювалося, що компоненти можуть спілкуватися один з одним, але тільки через виконавче середовище **COM**. У **COM** не було способу організувати пряму взаємодію компонентів, написаних на різних мовах, або створювати екземпляри один одного – усе це здійснювалося тільки за посередництва **COM**. Мало того, архітектура **COM** *не дозволяла реалізувати спадкування*, тим самим зводячи нанівець усі переваги об'єктно-орієнтованого програмування.

Ще одна пов'язана з цим проблема полягала в тому, що налаштовувати компоненти, написані на різних мовах, доводилося *незалежно один від одного*. Неможливо було у відлагоджувачі переходити від однієї мови до іншої. Тому насправді *під здатністю мовної взаємодії мається на увазі* можливість для класів, написаних на одній мові, безпосередньо звертатися до класів, написаних на іншій мові. Зокрема:

- клас, написаний на одній мові, може бути успадкований від класу, реалізованого на іншій мові;
- клас може містити екземпляр іншого класу, незалежно від того, на яких мовах написаний кожен з них;
- об'єкт може безпосередньо викликати методи іншого об'єкта, написаного на іншій мові;
- об'єкти (або посилання на об'єкти) можуть передаватися між методами;
- при виклику методів між мовами можна крокувати у відлагоджувачі за викликами, навіть, якщо це означає потребу переміщення між фрагментами початкового коду, написаними на різних мовах.

Усе це досить амбітні цілі, але як не дивно, завдяки **.NET** та **IL**, вони були досягнуті. Можливість переміщення між методами у відлагоджувачі *забезпечується інтегрованим середовищем розробки* (**IDE**) Visual Studio .NET, а не самому **CLR**.

1.7.2. Відмінність типів значень і типів посилань

Як і будь-яка мова програмування, **IL** пропонує множину наперед визначених примітивних типів даних. Одна з особливостей **IL**, проте, полягає в тому, що вона робить чітку *відмінність між типами значеннями* і *посилальними типами*. *Типи значень* – це ті, *змінні* які безпосередньо зберігають їх дані, тоді як *посилальні типи* – це ті, змінні які просто зберігають адресу, за якою відповідні дані можуть бути знайдені.

У термінах **C++** *посилальні типи* можна розглядати, неначе вони звертаються до змінних *через покажчик*, тоді як для **Visual Basic** краща аналогія для посилальних типів – це об'єкти, звернення до яких у **Visual Basic 6** завжди здійснюється за посиланням. У мові **IL** також встановлена своя специфікація відносно зберігання даних: *екземпляри посилальних типів* завжди *зберігаються* в області пам'яті, відомої як *керована куча* (managed heap), тоді як *типи значень* зазвичай *зберігаються у стеку* (хоча, якщо типи значень оголошені як поля всередині посилальних типів, то вони також будуть збережені у кучі).

1.7.3. Строга типізація даних

Один з найважливіших аспектів мови **IL** полягає в тому, що він базується на *виключно строгій типізації даних*. Це означає, що всі змінні мають чітко визначений конкретний тип даних (так, наприклад, в **IL** немає місця для типу даних **Variant**, який є у **Visual Basic** і сценарних мовах). Зокрема, **IL** зазвичай не допускає ніяких дій, які *дають у результаті невизначені типи даних*.

Наприклад, розробники **Visual Basic 6** можуть дозволити собі передавати змінні, не особливо піклуючись про їх типи, бо **Visual Basic** автоматично зробить потрібні перетворення. Розробники **C++** звикли до приведення покажчиків до різних типів. Можливість виконання таких дій корисна для продуктивності, але при цьому порушуються принципи безпеки типів. Тому такі дії допустимі лише

при деяких обставинах в деяких мовах, компільованих в керований код. Насправді, *покажчики* (на відміну від *посилань*) дозволені тільки у *позначених блоках коду* **C#** і зовсім не допускаються у **Visual Basic** (хоча і дозволені в керованому **C++**). Застосування покажчиків в коді приводить до того, що перевірки типів у пам'яті, виконувані **CLR**, завершуються невдало.

Варто відмітити, що *деякі мови, сумісні* з **.NET**, такі як **Visual Basic 2010**, все ще допускають деяку недбалість типізації, але це можливо лише тому, що компілятори, які знаходяться «за кулісами», гарантують безпеку типів у згенерованому коді **IL**.

Хоча забезпечення безпеки типів може спочатку привести до зниження продуктивності, все ж у багатьох випадках переваги, отримані від послуг, які надаються **.NET**, і які базуються на безпеці типів, набагато перевищують втрати від деякого зниження продуктивності.

Ці послуги включають наступне:

- можливість міжмовної взаємодії;
- прибирання сміття;
- безпека;
- домени застосувань.

У подальших розділах детально аналізується важливість строгої типізації даних для цих засобів **.NET**.

1.7.3.1. Строга типізація даних як ключ до міжмовної взаємодії

Якщо деякий клас успадкований від іншого або містить в собі екземпляри інших класів, то він повинен знати про всі типи даних, використовуваних іншими класами. Ось чому строга типізація даних така важлива. Дійсно, нестача інформації в будь-якій погодженій системі у минулому завжди була справжнім бар'єром для спадкоємство і взаємодії між різними мовами. Інформація подібного роду просто відсутня в стандартному виконуваному файлі або бібліотеці **DLL**. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

Нехай один з методів класу **Visual Basic 2010** визначений як такий, що повертає тип **Integer** – один із стандартних типів,

доступних у **Visual Basic 2010**. У мові **C#** *тип даних з таким ім'ям відсутній*. Зрозуміло, що здійснювати спадкування від цього класу, викликати цей метод і використати повернутий тип в коді C# можна тільки тоді, коли компілятор знатиме, як відобразити тип **Integer** мови **Visual Basic 2010** на деякий тип, визначений в **C#**. Яким чином ця проблема вирішується в **.NET**?

1.7.3.2. Загальна система типів

Ця проблема з типами даних вирішується в **.NET** за рахунок застосування *загальної системи типів* (**Common Type System – CTS**). Система **CTS** описує наперед обумовлені типи даних, які доступні в **IL**, тому усі мови, орієнтовані на середовище **.NET**, генерують скомпільований код, який зрештою базується на цих типах. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

Що стосується попереднього прикладу, то **Integer** з **Visual Basic 2010** – це насправді **32**-бітове *ціле зі знаком*, яке відображається безпосередньо на тип, відомий в **IL** як **Int32**. Тому це і буде той тип, який специфікований в коді **IL**. Через те, що компілятору **C#** відомо про цей тип, тут проблем немає. На рівні початкового коду **C#** посилається на **Int32** за ключовим словом **int**, тому компілятор просто буде трактувати цей метод **Visual Basic 2010** так, як ніби він повертає **int**.

CTS описує не просто примітивні типи даних, а цілу розвинену ієрархію типів, яка включає добре визначені точки, в яких код може визначати свої власні типи. Ієрархічна структура загальної системи типів (**CTS**) віддзеркалює об'єктно-орієнтовану методологію *одиночного спадкоємства* **IL** і показана на **рис. 1.6**.

У **C#** *кожен наперед визначений тип*, розпізнаний компілятором, відображається на один вбудований тип **IL**. Те ж справедливе і щодо **Visual Basic 2010**.

1.7.3.3. Загальна специфікація мови

Загальна специфікація мови (**Common Language Specification – CLS**) працює разом з **CTS** для забезпечення мовної взаємодії. **CLS** – це набір мінімальних стандартів, яких повинні дотримуватися усі компілятори, орієнтовані на **.NET**. Через те, що **IL** – дуже багата мова, розробники більшості компіляторів вважають за краще

обмежувати можливості конкретного компілятора підтримкою тільки підмножини засобів **IL** і **CTS**. Це нормально доти, поки компілятор підтримує все, що визначено в **CTS**. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

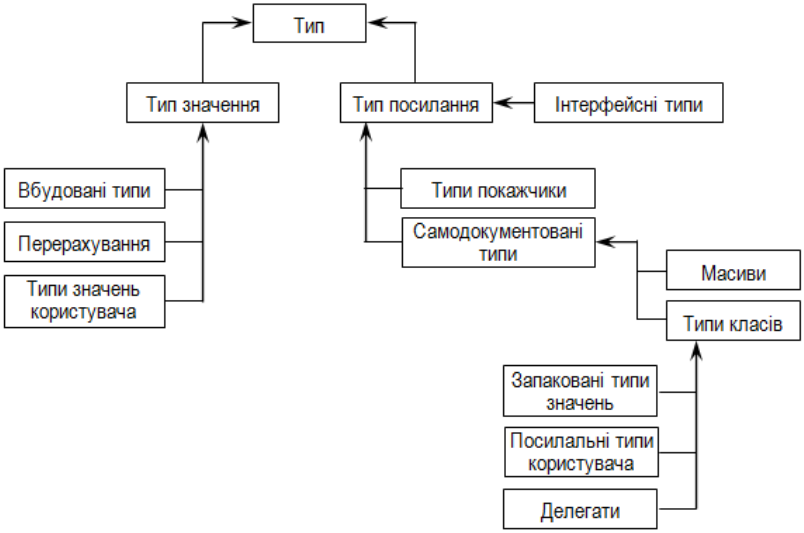


Рис. 1.6. Загальна система типів (**CTS**)

Наприклад, візьмемо чутливість до регістра. Мова **IL** чутлива до регістра символів. Розробники, які пишуть на мовах, чутливих до регістра, широко використовують гнучкість, яку забезпечує ця залежність від регістра, при виборі імен змінних. Проте мова **Visual Basic 2010** не чутлива до регістра символів. Специфікація **CLS** обходить цю проблему зазначаючи, що будь-який **CLS**-сумісний код не повинен включати ніяких пар імен, які відрізняються тільки регістром символів. Отже, код **Visual Basic 2010** може працювати з **CLS**-сумісним кодом.

Цей приклад ілюструє, що **CLS** працює двома способами:

1. Індивідуальні компілятори недостатньо потужні, щоб підтримувати всі можливості **.NET** – це є складність для

розробників компіляторів інших мов програмування, націлених на **.NET**.
2. Якщо обмежити класи лише **CLS**-сумісними засобами, то код, написаний на будь-якій іншій мові програмування, зможе користуватися цими класами.

Витонченість цієї ідеї полягає в тому, що *обмеження* у використанні **CLS**-сумісних засобів *накладаються* тільки *на загальнодоступні, захищені члени класів* і *на загальнодоступні класи*. У межах *приватної реалізації класів* ви вільні писати будь-який несумісний з **CLS** код, бо код з інших збірок (одиниць керованого коду – див. далі у цьому розділі) у будь-якому випадку не має доступу до цієї частини коду.

Ми тут не занурюватимемося у подробиці специфікації **CLS**. У загальному випадку **CLS** не особливо зачіпає код **C#**, бо в **C#** є не так вже багато засобів, не сумісних з **CLS**.

Цілком допустимо писати код, не сумісний з **CLS**. Проте у цьому випадку немає гарантії, що скомпільований **IL**-код буде повністю здатним до взаємодії.

1.7.3.4. Прибирання сміття

Прибиральник сміття (garbage collector) – це відповідь **.NET** на питання управління пам'яттю, зокрема, на питання про те, що робити з відновленням пам'яті, яку просить застосування. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти. До теперішнього часу на платформі Windows використовувалися дві технології звільнення пам'яті, виділеної системою за динамічними запитами:

1. Ручне виконання цього завдання в коді застосування.
2. Підтримка лічильників посилань на об'єкти.

Покладання відповідальності за звільнення пам'яті на код застосування – це техніка, використовувана в низькорівневих високопродуктивних мовах на кшталт **C++**. Це ефективно і має ту перевагу, що (у загальному випадку) ресурси ніколи не бувають зайнятими довше, ніж це треба. Проте *значний недолік* такого підходу полягає в тому, що він часто породжує помилки. Код, який

просить пам'ять, також зобов'язаний явно інформувати систему, коли потреба у ній відпадає. Про це дуже легко забути, що приведе до витоків пам'яті.

Хоча сучасні середовища розробки пропонують інструменти, які допомагають виявити такі витоки, все ж вони пропускають помилки, які дуже важко відстежити, бо вони ніяк не проявляються доти, поки не витече настільки багато пам'яті, що Windows відмовить процесу в розподілі нової. І в такий момент увесь комп'ютер почне помітно «*гальмувати»* при кожному зверненні до пам'яті.

Підтримка лічильників посилань – це техніка, використовувана в **COM**. Ідея полягає в тому, що кожен компонент **COM** *підтримує лічильник клієнтів*, які в даний момент посилаються на нього. Коли *значення лічильника падає до нуля*, компонент може знищити себе і звільнити асоційовану з ним пам'ять і ресурси. При такому підході проблема зводиться до обов'язку клієнта «коректно поводитися» і вчасно сповіщати компонент про те, що він завершив з ним працювати. *Варто одному клієнтові не зробити цього*, і об'єкт залишиться в пам'яті. Іноді ця проблема потенційно серйозніша, ніж простий витік пам'яті в **C++**, тому що **COM**-*компонент може існувати всередині свого власного процесу*, а це означає, що він ніколи не буде видалений системою (по крайній мірі, що стосується витоків пам'яті **C++**, то система може запросити назад всю пам'ять процесу, коли він завершиться).

Замість усього цього **.NET** *покладається на прибиральника сміття*. Ця програма, призначена для того, щоб *очищати пам'ять*. Ідея полягає в тому, що вся пам'ять, яка динамічно виділяється розподіляється в області кучі (це правильно для всіх мов, хоча у випадку **.NET CLR** *підтримує власну керовану кучу* для використання застосуваннями **.NET**). Кожен раз, коли **.NET** виявляє, що керована куча цього процесу заповнилася і потребує упорядковування, він викликає прибиральник сміття. Цей прибиральник сміття переглядає всі змінні, які знаходяться в контексті вашого коду, і перевіряє посилання на об'єкти, розташовані в області кучі, щоб ідентифікувати, які з них доступні з коду, тобто, *довідатися, на які об'єкти існують посилання*. Будь-який об'єкт з відсутністю посилань на нього розглядається як такий, до якого більше не буде здійснюватися доступ з коду, тому об'єкт

може бути видалений. Подібний механізм прибирання сміття застосовується і в **Java**.

Прибирання сміття працює в **.NET**, тому що **IL** спроектований так, щоб полегшити цей процес. Принципи організації **IL** свідчать, що *отримати посилання на існуючий об'єкт* не можна інакше, окрім як копіюванням існуючих посилань, а також те, що **IL** – мова безпечна до типів. У цьому контексті мається на увазі, що якщо існує якесь посилання на об'єкт, то в цьому посиланні міститься інформація, достатня для того, щоб точно визначити тип об'єкта.

Механізм прибирання сміття неможливо застосовувати з такою мовою, як, наприклад, *некерований* **C++**, тому що **C++** допускає вільне приведення покажчиків до інших типів. Одна важлива властивість прибирання сміття полягає в тому, що це *не детермінований процес*. Іншими словами, неможливо довідатися, коли буде викликаний прибиральник сміття; він буде викликаний тоді, коли середовище **CLR** вирішить, що в цьому є потреба, хоча можна перевизначити цей процес і явно викликати прибиральник сміття в коді застосування.

1.7.3.5. Безпека

*Механізм безпеки***. NET** перевершує механізми безпеки, які надаються **Windows**, *бо забезпечує захист на рівні коду*, тоді як **Windows** *насправді забезпечує тільки безпеку на рівні ролей*. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

Безпека, яка базується на ролях (role-based security), базується на ідентифікації облікового запису, під яким працює процес (тобто, хто запускає процес і володіє ним). *Безпека на рівні коду*, з іншого боку, базується на тому, *що* саме робить код, і наскільки йому можна довіряти. Завдяки строгій типізації **IL**, **CLR** може інспектувати код перед його запуском з тим, щоб визначити потрібні повноваження безпеки. **.NET** також пропонує механізм, за допомогою якого код може завчасно запросити потрібні для роботи привілеї.

Важливість *безпеки, яка базується на коді* (code-based security), в тому, що вона *знижує ризик*, пов'язаний з кодом сумнівного походження (такого, яким може бути код, завантажений з Інтернету). Наприклад, навіть якщо код виконується від імені облікового запису адміністратора, залишається можливість за допомогою захисту рівня

коду вказати, що цьому коду не дозволено виконувати дії певного роду, які зазвичай дозволені привілейованому користувачеві (адміністраторові), наприклад, читання і запис змінних середовища, читання і запис реєстру або доступ до засобів рефлексії **.NET**.

1.7.3.6. Домени застосувань

Домени застосувань (application domains) – важливе нововведення **.NET**, призначене для *зниження накладних витрат*, пов'язаних із запуском застосувань, які *мають бути ізольовані один від одного*, але *при цьому потребують взаємодії між собою*. Класичний приклад цього – застосування веб-сервера, які можуть спільно відповідати на багато запитів браузерів, а тому повинні, ймовірно, мати екземпляри компонента, відповідального за паралельне обслуговування таких запитів. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

У часи, які передували появі **.NET**, доводилося вибирати між тим, щоб *дозволити цим екземплярам розділяти один і той же процес* (у результаті ризикуючи припиненням роботи усього веб-сайту у разі виникнення проблем з одним екземпляром, який виконується), та ізолюванням цих екземплярів в окремих процесах, що приводило до росту накладних витрат і зниження продуктивності.

До останнього часу *ізоляція коду була можлива тільки між процесами*. Коли запускається нове застосування, воно працює в контексті процесу. **Windows** *ізолює* процеси один від одного за допомогою адресних просторів. Ідея полягає в тому, що кожному процесу надається **4** Гбайт *віртуальної пам'яті*, в якій він зберігає свої дані і виконуваний код (**4** Гбайт – це для **32**-розрядних систем, а **64**-розрядні системи використовують більший обсяг пам'яті). **Windows** вводить додатковий рівень переадресації, завдяки якому ця віртуальна пам'ять відображається на визначену область реальної фізичної пам'яті або дисковий простір. Кожен процес отримує своє відображення, при цьому гарантується, що не станеться ніякого перекриття між реальними фізичними областями пам'яті, виділеними кожному з них (**рис. 1.7**).

У загальному випадку будь-який процес може звернутися до пам'яті тільки за адресою віртуальній пам'яті – *процеси не мають прямого доступу до фізичної пам'яті*. А тому просто неможливо одному процесу отримати доступ до пам'яті, виділеної для іншого.

Це гарантує, що ніякий код, який «погано» поводиться, не зможе пошкодити нічого поза власним адресним простором. Потрібно зазначити, що у **Windows 95/98** цей захист ще не був такий досконалий, як в **Windows NT** та пізніших версіях, тому у старих системах існувала теоретична можливість для застосувань порушити роботу **Windows**, виконавши запис в недозволену область пам'яті.

Але процеси не лише служать для ізоляції один від одного екземплярів працюючого коду. У системах **Windows NT** і пізніших вони також формують елементи, яким призначаються права доступу і привілеї безпеки. Кожен процес забезпечений власним *маркером доступу* (security token), який вказує системі, які саме дії дозволено виконувати цьому процесу.

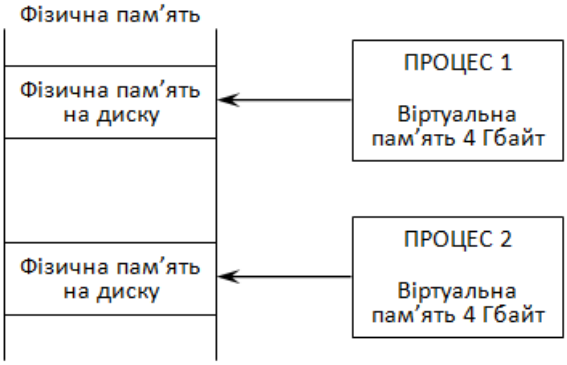


Рис. 1.7. Відображення віртуальної пам'яті процесів на фізичну пам'ять

Хоча процеси зручні для забезпечення безпеки, все ж вони мають серйозний недолік, пов'язаний з продуктивністю. Часто буває так, що багато процесів працюють разом і тому потребують взаємодії один з одним. Очевидний приклад цього – коли процес звертається до **COM**-компонента, який є виконуваним, а тому запускає свій власний процес. Те ж трапляється в **COM**, коли використовується «*заступник*» (surrogate). Через те, що процес не може розділити пам'ять з іншими, має бути застосований складний *процес маршалізації* для копіювання даних між процесами. У результаті відчутно страждає продуктивність. Якщо потрібно, щоб компоненти працювали разом без зниження продуктивності, то треба

використати компоненти, які базуються на **DLL**, щоб все працювало в одному адресному просторі, ризикуючи, що один компонент, який «погано» себе поводить зруйнує всі інші.

Домени застосувань спроектовані як спосіб розділення компонентів без супутніх проблем зниження продуктивності, пов'язаних з передачею даних між процесами. Ідея полягає в тому, що *кожен процес розділяється на декілька доменів застосувань*. Кожен домен застосування приблизно відповідає окремому застосуванню, і кожен потік виконання запускається у певному домені застосування (**рис. 1.8**).

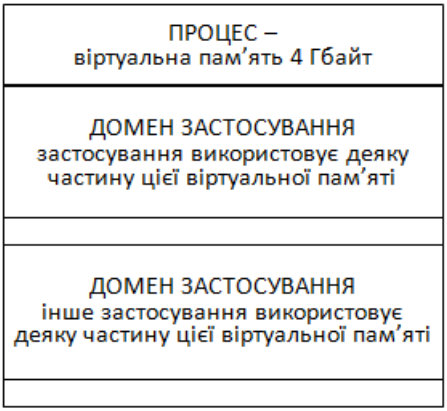


Рис. 1.8. Домени застосувань

Якщо різні виконувані програми запускаються в одному адресному просторі процесу, то цілком зрозуміло, що вони можуть легко розділяти спільні дані, тому що теоретично в змозі безпосередньо бачити дані один одного. Проте, хоча така можливість існує в принципі, **CLR** забезпечує, щоб це не траплялося на практиці. Для цього **CLR** інспектує код кожного працюючого застосування, гарантуючи, що код не вийде за межі області своїх власних даних. На перший погляд цей трюк здається майже неможливим – врешті-решт, як можна довідатися, що збирається робити програма, до того, як вона буде запущена?

Насправді, зазвичай, це можна зробити завдяки строгому захисту типів **IL**. У більшості випадків, якщо тільки код не

використовує *небезпечних засобів*, таких як *покажчики*, то використовувані ним типи даних гарантують, що не станеться ніяких неправильних звертань до пам'яті.

Наприклад, типи масивів **.NET** передбачають перевірку меж, щоб не допустити ніяких дій за межами масиву. Якщо функціонуюче застосування потребує взаємодії або розділення даних з іншим працюючим застосуванням, воно повинно робити це за допомогою виклику віддалених служб.

Код, *верифікований* на предмет того, що він *не може звертатися до пам'яті поза межами домена застосування* (окрім як явним використанням віддалених механізмів), називають *безпечним до типів у пам'яті* (memory type-safe). Такий код може бути безпечно запущений паралельно з іншим безпечним до типів кодом, у різних доменах застосувань у рамках одного процесу.

1.8. Обробка помилок за допомогою виключень

Середовище **.NET** спроектоване для полегшення обробки помилкових ситуацій на основі механізму, який використовує *виключення*, який використовується також у мовах **Java** і **C++**. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти. Розробники **C++** повинні відмітити, що через те, що **IL** – система, яка строго типізується, у ній не відбувається ніякого зниження продуктивності, пов'язаного із застосуванням виключень, як це має місце в **C++**. До того ж *блок* **finally**, про потребу у якому часто пишуть розробники **C++**, в **.NET** і **C#** підтримується з самого початку.

Коротко кажучи, ідея полягає в тому, що *деякі області коду можуть бути помічені як процедури обробки виключень*, кожна з яких може обробити певну помилкову ситуацію (наприклад, відсутність файлу або відмова в доступі до якоїсь дії). *Архітектура виключень гарантує*, що при виникненні помилкових умов виконання негайно перейде до потрібної процедури-обробника, яка найкраще годиться для обробки цієї умови виключення.

Архітектура обробки виключень також надає узгоджену можливість передачі об'єктів процедурі-обробнику. Цей об'єкт може включати відповідне повідомлення користувачу і подробиці щодо того, де саме в коді було виявлено виключення.

Велика частина архітектури обробки виключень, включаючи потік управління програми при виникненні виняткових ситуацій, *обробляється мовами високого рівня* (C#, Visual Basic 2010, C++) і *не підтримується ніякими спеціальними командами* **IL**. Так, наприклад, **C#** обробляє виключення, використовуючи блоки коду **try { }, catch { }** і **finally { }**.

Що робить **.NET**, так це надає інфраструктуру, яка дозволяє компіляторам, орієнтованим на **.NET**, підтримувати обробку помилок. Зокрема, вона пропонує набір класів, які представляють виключення, а також забезпечує мовну взаємодію, яка дозволяє обробляти об'єкти згенерованих виключень в обробниках, написаних на мовах, незалежних від мови, на якій реалізований код, який згенерував виключення. Подібна незалежність від мови недосяжна ні в реалізації виключень **C++**, ні в реалізації **Java**, хоча і присутня в обмеженому розширенні механізму **COM**, який обробляє помилки, і який включає повернення кодів помилок з методів і передачу об'єктів помилки назовні. Факт *погодженої обробки виключень* у різних мовах – ключовий аспект гнучкості багатомовної розробки.

1.9. Застосування атрибутів

Атрибути (attribute) – цей засіб, відомий розробникам **C++**, які займаються створенням **COM**-компонентів (завдяки використанню мови визначення інтерфейсу **COM** – **Microsoft's COM Interface Definition Language**, або, коротко, **IDL**). Первинна ідея атрибуту полягала в тому, щоб надати додаткову інформацію щодо визначеного елемента програми, який може бути використаний компілятором. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

Атрибути включені в **.NET** і тому тепер підтримуються в **C++, C#, Visual Basic 2010**. Новим у застосуванні атрибутів **.NET**, є та обставина, що тепер *можна визначати власні атрибути*, які налаштовуються, в початковому коді. Ці визначені користувачами атрибути поміщаються в метадані, які супроводжують типи даних і методи програми. Це може виявитися зручним для цілей документування, коли атрибути можуть застосовуватися спільно з технологією рефлексії для вирішення завдань програмістів, яка базується на атрибутах. До того ж, у відповідності з філософією мовної взаємодії **.NET**, атрибути можуть бути визначені в коді,

написаному на одній мові, і прочитані в коді, написаному на іншій мові.

1.10. Збірки

Збірка (assembly) – це логічна одиниця, яка містить скомпільований код для **.NET Framework**. Тут ми не розглядаємо їх надто детально, а лише говоримо про деякі основні моменти.

Збірка – це повністю самодостатній і *швидше логічний, ніж фізичний елемент*. Це означає, що він може бути збережений у більш ніж одному файлі (хоча динамічні збірки зберігаються в пам'яті, а зовсім не у файлах). Якщо збірка зберігається у більше ніж одному файлі, то повинен існувати один головний файл, який містить точку входження і який описує інші файли.

Треба зазначити, що одна і та ж структура збірки використовується як для виконуваного коду, так і для коду бібліотек. Єдина реальна відмінність *виконуваної збірки* полягає в тому, що вона містить головну точку входження програми, тоді як *бібліотечна збірка* – ні.

Важлива властивість збірок полягає в тому, що вони *містять метадані*, які описують типи і методи, визначені в її коді. Окрім цього *збірка зберігає в собі метадані*, які описують її саму. Ці метадані, які зберігаються в *області маніфесту*, дозволяють виконувати перевірку номера *версії збірки* та *її цілісність*. Для інспекції вмісту збірки, включаючи маніфест і метадані, може використовуватися Windows-утиліта **ildasm**.

Той факт, що збірка містить метадані програми, означає, що застосування або інші збірки, які викликають код даної програми, не потребують звернення до реєстру або будь-якого іншого джерела даних, щоб довідатися, як конкретну збірку треба використати. Це істотний прорив у порівнянні із старим способом роботи **COM**, коли **GUID**-ідентифікатори компонентів та інтерфейсів треба було витягати з реєстру, а подробиці методів і властивостей в деяких випадках читати з бібліотеки типів.

Зберігання даних в трьох різних місцях приводило до очевидного ризику отримати несинхронізовані частини, що виключало використання цього компонента іншим програмним забезпеченням. Що стосується збірок, то тут ризик подібного роду виключений, тому що всі метадані зберігаються разом з

виконуваними інструкціями програми. Треба зазначити, що попри те, що збірки можуть зберігатися в декількох файлах, проблема синхронізації не виникає, бо в тому ж файлі, де знаходиться точка входження, зберігається й інформація про вміст інших файлів. Тобто навіть якщо один з них підмінити або зіпсувати, то це негайно буде виявлено, і всій збірці буде заборонено завантажуватися на виконання. Збірки бувають двох видів: *розділювані* і *приватні*.

1.10.1. Приватні збірки

Це *простий тип збірок*. Зазвичай вони постачаються з певним програмним забезпеченням і призначені для застосування тільки в його складі. Звичайний сценарій отримання приватної збірки – це коли застосування постачається у вигляді виконуваної програми і багатьох бібліотек, код яких може бути використаний тільки цим застосуванням.

Система гарантує, що приватні збірки не будуть використовуватися іншим програмним забезпеченням, тому що застосування може завантажувати тільки приватні збірки, які знаходяться в тій же теці, де зберігається і головна виконувана програма, яка їх завантажує, або у вкладених теках.

Через те, що зазвичай можна розраховувати на те, що комерційне програмне забезпечення завжди встановлюється у власний каталог, то відсутня небезпека того, що один програмний пакет перепише, модифікує або ненавмисне завантажить приватні збірки, які належать іншому пакету. Через те, що приватні збірки можуть бути використані тільки тим пакетом програмного забезпечення, для якого вони призначені, є *можливість управляти* тим програмним забезпеченням, яке їх використовує. Звідси зменшується потреба у посиленні заходів безпеки, бо немає ризику, наприклад, того, що якесь комерційне програмне забезпечення перепише збірки їх новими версіями (за винятком випадків, коли програмне забезпечення спеціально розроблене для нанесення умисної шкоди). Не існує також *проблеми колізії імен*. Якщо виявиться, що класи в приватній збірці мають ті ж імена, що і класи в іншій чужій приватній збірці, то це немає значення, бо кожне застосування буде в змозі бачити тільки один набір власних приватних збірок.

Через те, що приватна збірка повністю самодостатня, процес її розгортання дуже простий. Ви просто поміщаєте відповідний файл (чи файли) у відповідну теку системи (ніяких записів вносити в реєстр не вимагається). Цей процес відомий як *встановлення з нульовою дією* або встановлення за допомогою **xcopy** (zero impact (xcopy) installation).

1.10.2. Розділювані збірки

Призначення розділюваних збірок – бути *бібліотеками загального використання*, які можуть використовуватися будь-яким іншим застосуванням. Через те, що будь-яке інше застосування може отримати доступ до розділюваної збірки, то треба бути обережним, щоб виключити описані нижче риски.

- Колізія імен, коли розділювана збірка, постачається іншою компанією, реалізує типи з тими ж іменами, які використовуються у вашій збірці. Через те, що клієнтський код теоретично може мати доступ до двох таких збірок одночасно, це може бути серйозною проблемою.
- Ризик того, що ця збірка буде перезаписана іншою версією тієї ж збірки, і нова версія виявиться несумісною з деяким існуючим клієнтським кодом.

Рішення цих проблем включає *розміщення розділюваних збірок* у спеціальному піддереві каталогів файлової системи, відомому під назвою *глобальний кеш збірок* (global assembly cache – **GAC**). На відміну від приватних збірок, це не може бути зроблено простим копіюванням її у певну теку – збірку знадобиться спеціальним чином встановити в кеші **GAC**. Цей процес може бути реалізований з допомогою багатьох утиліт **.NET** і включає виконання визначених перевірок встановлюваної збірки, а також створення невеликої ієрархії тек в межах **GAC**, використовуваних для забезпечення цілісності збірок.

Щоб *виключити колізії імен*, розділюваним збіркам, призначаються імена, які базуються на *шифруванні індивідуальним ключем* (на відміну від приватних збірок, яким дається ім’я, яке співпадає з ім’ям головного файла). Це ім’я, відоме як *строге ім’я*,

гарантовано *є унікальним*, і воно повинне вказуватися застосуванням, яке посилається на цю розділювану збірку.

Проблеми, пов'язані *з ризиком перезапису збірки*, вирішуються зазначенням інформації про версію у *маніфесті збірки*, а також можливістю паралельних встановлень різних версій цієї збірки.

1.10.3. Рефлексія

Через те, що збірки зберігають в собі метадані, включаючи подробиці будови типів і членів цих типів, визначених у збірці, *існує можливість отримання програмного доступу до цих метаданих*. Тут ми не розглядаємо це детально, а лише говоримо про деякі основні моменти. Ця технологія, яка називається *рефлексією* (reflection), відкриває цікаві можливості, бо означає, що керований код може досліджувати інший керований код або навіть сам себе, щоб витягнути інформацію про код. Найчастіше це використовується для отримання подробиць про атрибути, хоча також, окрім іншого, рефлексію можна застосовувати як непрямий спосіб створення екземплярів класів чи виклику методів, надаючи їх імена у вигляді рядків. Отже, на основі введення користувача можна під час виконання, а не під час компіляції вибирати класи для створення їх екземплярів і методи для виклику (це називається *динамічним зв'язуванням*).

1.10.4. Паралельне програмування

У .NET Framework 4.0 з'явилася можливість використати переваги сучасних *процесорів з багатьма ядрами*. Нові засоби паралельних обчислень пропонують засоби розділення робочих дій і виконання їх на декількох процесорах. Доступні сьогодні нові **API**-інтерфейси паралельного програмування значно спростили написання безпечного багатопотокового коду, проте важливо розуміти, що вони не позбавляють від ситуацій змагань, а також від таких речей, як блокування.

Нові засоби паралельного програмування надаються *бібліотекою* **Task Parallel Library**, а також механізмом **PLINQ Execution Engine**. Для з'ясування суті паралельного програмування треба це питання вивчати окремо і значно детальніше.

1.11. Класи .NET Framework

Можливо, найбільша перевага написання керованого коду – у крайньому разі, з погляду розробника – полягає в тому, що ви отримуєте можливість використати *бібліотеку базових класів* **.NET**.

Базові класи **.NET** надають величезну колекцію класів керованого коду, які дозволяють вирішувати практично будь-які завдання, які раніше можна було вирішувати з допомогою **Windows API**. Усі ці класи спадкують ту ж об'єктну модель **IL** з *одиночним спадкоємством*. Це означає, що можна або *створювати об'єкти* будь-якого з базових класів **.NET**, або *спадкувати* від них власні класи.

*Відмінність базових класів***. NET** полягає в тому, що вони спроектовані інтуїтивно зрозумілими і простими у використанні. Наприклад, для *запуску потоку* треба викликати метод **Start()** класу **Thread**. Щоб *зробити недоступним* об'єкт **TextBox**, *властивості* **Enabled** цього об'єкта присвоюється значення **false**. Такий підхід, добре знайомий розробникам **Visual Basic** і **Java**, чиї бібліотеки використати так само легко, і це принесе величезне полегшення розробникам **C++**, яким впродовж багатьох років доводилося «воювати» з такими **API**-функціями, як **GetDIBits(), RegisterWndClassEx()** та **IsEqualIID()**, а також з множиною функцій, які вимагали передачі дескрипторів вікон.

Проте розробники на **C++** завжди мали легкий доступ до повного набору **Windows API**, тоді як розробники на **Visual Basic 6** і **Java** *були обмежені у використанні базової функціональності операційної системи*, доступ до якої вони отримували зі своїх мов. Що стосується базових класів **.NET**, то вони комбінують простоту використання, властиву бібліотекам **Visual Basic** і **Java**, з *відносно повним покриттям набору функцій* **Windows API**. Багато засобів **Windows** *не доступні через базові класи*, і в цих випадках доведеться звертатися до **API**-функцій, але, загалом, це стосується лише найекзотичніших функцій. Для повсякденного застосування набору базових класів, в основному, буде достатньо. Але якщо знадобиться викликати **API**-функцію, то для цього **.NET** надає так званий *механізм виклику платформи* (platform-invoke), який гарантує коректне перетворення типів даних, тому тепер це завдання не складніше, ніж виклик цих функцій безпосередньо з коду **C++**,

причому незалежно від того, якою мовою пишеться код – **C#, C++** або **Visual Basic 2010**.

Для перегляду *класів, структур, інтерфейсів* і *перерахувань* базової бібліотеки класів може застосовуватися **Windows**-утиліта **WinCV**.

1.12. Простори імен

Простори імен (namespace) – це спосіб, завдяки якому **.NET** уникає *конфліктів* імен між класами. Вони призначені для того, щоб виключити ситуації, коли ви визначаєте клас, який представляє замовника, називаєте його **Customer**, а після цього хтось інший робить те ж саме (подібний сценарій досить поширений).

Простір імен – це не більше ніж група типів даних, але яка дає той ефект, що імена усіх типів даних в межах простору імен автоматично *забезпечуються префіксом – назвою простору імен*. Простори імен можна вкладати один в інший. Наприклад, *більшість базових класів* **.NET** загального призначення знаходяться в *просторі імен* **System**. Базовий клас **Array** відноситься до цього простору, тому його *повне ім'я* – **System.Array**.

Платформа **.NET** вимагає, щоб всі імена були оголошені в межах простору імен, наприклад, ви можете помістити свій клас **Customer** в *простір імен***YourCompanyName**. Тоді *повне ім'я цього класу* виглядатиме як **YourCompanyName.Customer**.

Якщо простір імен не вказаний явно, тип буде доданий до безіменного глобального простору імен.

У більшості ситуацій Microsoft рекомендує застосовувати хоч би два вкладених простори імен: *перший* – найменування вашої компанії, а *другий* – назва технології чи пакету ПЗ, до якого відноситься клас, щоб це виглядало приблизно так: **YourCompanyName.SalesServices.Customer**. У більшості випадків такий підхід захистить класи вашого застосування від можливих конфліктів з іменами класів, написаних розробниками з інших компаній.

РОЗДІЛ 2. ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ MICROSOFT VISUAL STUDIO .NET

Microsoft Visual Studio .NET – це інтегроване середовище розробки **IDE** (Integrated Development Envirionment) для створення, запуску і налаштування застосувань на різних мовах програмування .NET. Ми здійснимо огляд Visual Studio 2013 IDE та опишемо процес створення простого застосування Visual C#, яке базується на розміщенні готових структурних елементів у потрібних місцях – це називається **візуальною розробкою застосувань**.

2.1. Огляд Visual Studio 2013 IDE

Visual Studio існує у декількох версіях. У більшості прикладів ми будемо використовувати **Visual Studio Ultimate 2013**. Надалі ми називатимемо **Visual Studio Ultimate 2013 IDE** просто **Visual Studio** або **VS**, або **IDE**.

Для позначення команд меню використовуватимемо знак **|**. Наприклад, запис **File | Open File...** означає, що треба вибрати команду **Open File...** з меню **File**. IDE запускається командою **Пуск | Все программы | Microsoft Visual Studio 2013**. На початку виконання виводиться початкова сторінка (**рис. 2.1**). Залежно від версії **Visual Studio** ця сторінка може виглядати інакше.

На початковій сторінці знаходиться список посилань на ресурси Visual Studio і на ресурси в Інтернеті. До початкової сторінки можна у будь-який момент повернутися командою **View | Start Page**.

2.1.1. Посилання на початковій сторінці

Посилання на початковій сторінці виводяться у два стовпці. Посилання з розділу **Start** лівого стовпця дозволяють перейти до створення нових застосувань або продовжити роботу над існуючим проектом. У розділі **Recent (Последние)** містяться посилання на нещодавно створені або змінені проекти. Створювати нові або відкривати існуючі проекти також можна за допомогою посилань з розділу **Start**.

Правий стовпчик складається з двох вкладок – **Get Started (Начало работы –** вибрана за замовчуванням**)** і **Latest News (Последние новости)**. Посилання на вкладці **Get Started** надають

інформацію про мови програмування, підтримувані Visual Studio, і різні освітні ресурси. Для звернення до цієї інформації з IDE потрібне під'єднання до Інтернету.

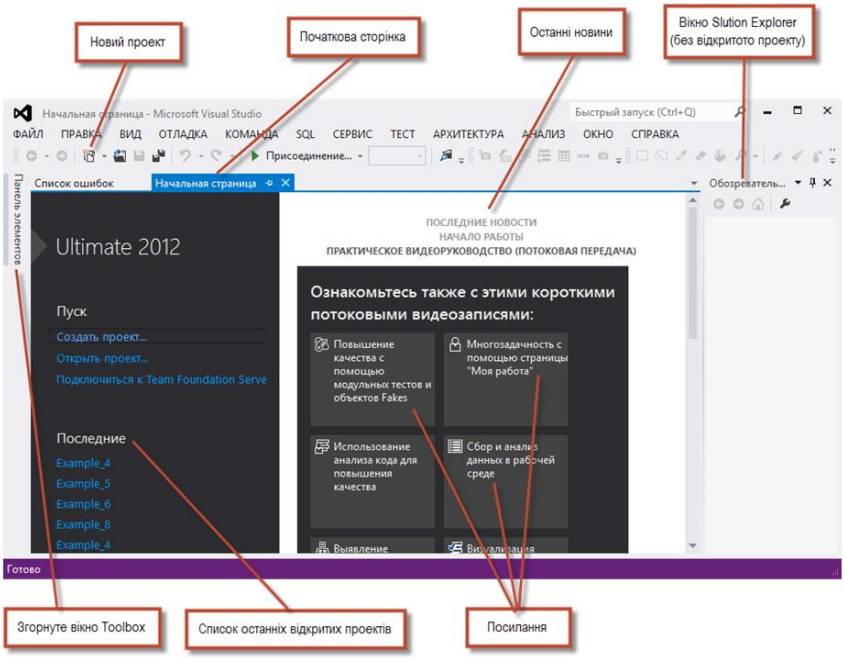


Рис. 2.1. Початкова сторінка у Visual Studio Ultimate 2013

На вкладці **Latest News** знаходиться кнопка **Enable RSS Feed**. Якщо клацнути на ній, IDE виводить посилання на останні новини щодо Visual Studio (наприклад, оновлення і виправлення помилок) та інформацію з нетривіальних питань розробки. Детальнішу інформацію щодо Visual Studio можна знайти в **MSDN** (Microsoft Developer Network) Library за адресою **msdn.microsoft.com/en-us/library/default.aspx**.

На сайті MSDN зібрані статті, завантажувані матеріали і підручники з технологій, які можуть бути цікавими для розробників Visual C#. Ви також можете переглядати веб-сторінки з IDE командою **View | Other Windows | Web Browser**. Щоб звернутися до веб-сторінки, введіть її **URL**-адресу в адресному рядку (**рис. 2.2**) і

натисніть клавішу **Enter**. Веб-сторінка, яка переглядається, відкривається в новій вкладці в IDE (див. **рис. 2.2**).

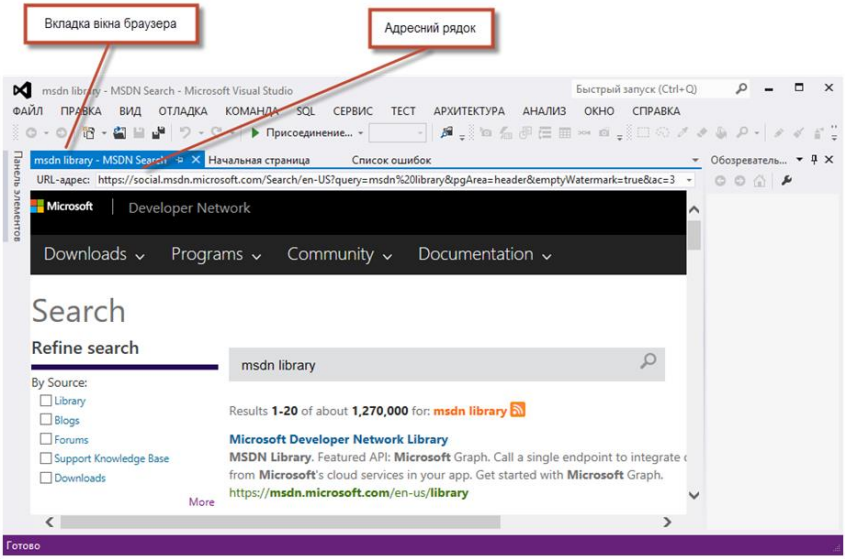


Рис. 2.2. Веб-сторінка MSDN Library у Visual Studio

2.1.2. Створення нового проекту

Розробка застосування у Visual C# розпочинається зі створення нового або відкриття існуючого проекту. Щоб створити новий проект, треба виконати команду **File | NewProject...**, а щоб відкрити існуючий – команду **File | OpenProject...**. Також можна клацнути на посиланнях **New Project...** або **Open Project...** у розділі **Start**. Проект є групою взаємозв'язаних файлів (наприклад, які містять код Visual C# або графіку, використовувану у застосуванні). У Visual C# застосування організовуються в **проекти** (projects) і **рішення** (solutions), які містять один або декілька проектів. Рішення використовуються при створенні великомасштабних застосувань. Більшість застосувань, розроблюваних нами, складаються з рішення з одним проектом.

2.2. Види проектів

Visual Studio .NET для мов C#, Visual Basic і J# пропонує різноманітні види проектів (див. **рис. 2.3** на якому представлені проекти для мови C#). Серед них є порожній проект, в якому спочатку не міститься ніякої функціональності; є також проект, орієнтований на створення **Web**-служб.

У лівій частині цієї діалогової панелі можна вибрати тип проекту. У загальному випадку можна вибрати проекти, створені на мовах програмування Visual Basic .NET, C#, C++, а також на деяких інших мовах. Цей список залежить від того, які мови були вибрані при встановленні Visual Studio, а також від того, чи були придбані і встановлені додаткові мови програмування сторонніх виробників.

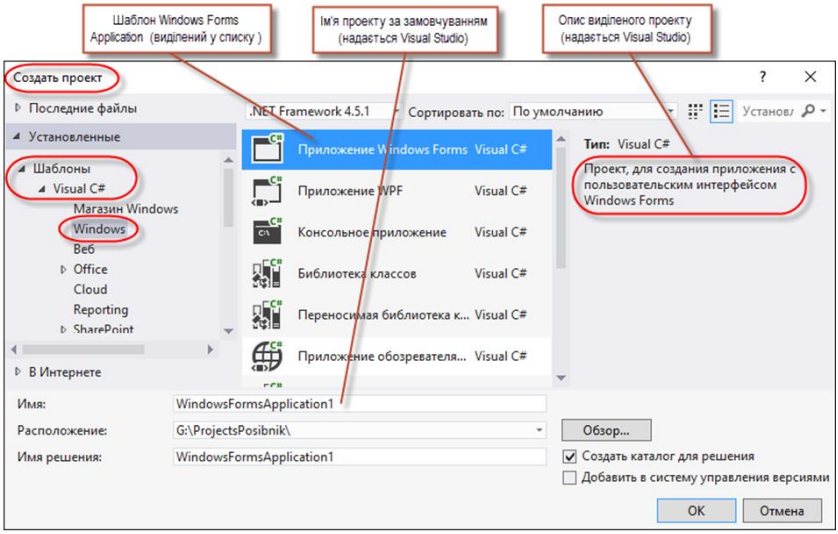


Рис. 2.3. Пропоновані проекти для мови C#

У правій частині екрану можна вибрати один із запропонованих шаблонів для даного типу проектів:

- **Windows Application (Приложения Windows Forms)** – шаблон для Windows-застосування. Застосування **Windows Forms** виконується в операційній системі Windows і

зазвичай має **графічний інтерфейс** (**GUI**), з яким взаємодіють користувачі;

- **Windows Presentation Foundation (Приложения WPF)** – підсистема для побудови графічних інтерфейсів;
- **Console Application (Консольное приложение)** – шаблон для створення консольних застосувань;
- **Class Library (Библиотека классов)** – шаблон для створення бібліотеки класів, яка буде використовуватися іншими застосуваннями;
- **Portable Class Library (Переносимая библиотека классов) –** шаблон переносної бібліотеки класів, який дозволяє писати працюючі на декількох платформах .NET Framework *керовані збірки* і виконувати їх побудову. Можна створювати класи, які містять код, який треба використати у багатьох проектах (наприклад, код спільно використовуваної бізнес-логіки), і потім посилатися на ці класи з проектів різних типів;
- **WPFBrowser Application (Приложение обозревателя WPF)** – це WPF-застосування, яке виконується всередині браузера Web-сторінок. Такі застосування є повноцінними застосуваннями WPF з деякими відмінностями;
- **WPF Custom ControlLibrary(Библиотека настраиваемых элементов управления WPF)** – шаблон WPF, який входить до складу Visual Studio дозволяє створювати **Windows Presentation Foundation** (WPF) елементи управління, які автоматично додаються в панелі інструментів при встановленні розширення;
- **WPF User Control Library(Библиотека пользовательских элементовуправления WPF)** – шаблон елемента управління панелі елементів WPF (**Windows Presentation Framework**), який дозволяє створювати елементи управління WPF, що автоматично додаються до елементів управління при встановленні розширення;
- **Empty Project/Empty Web Project (Пустой проект)** – проект, який створюється без використання шаблонів;
- **Windows Service (Служба Windows)** – шаблон для створення сервісів ОС;

- **Windows Forms Control Library (Библиотека элементов управления Windows Forms)** – Шаблон проекту «**Библиотека элементов управления Windows»** призначений для створення нестандартних елементів управління, використовуваних у формах Windows Forms. Шаблон автоматично додає найважливіші посилання і файли проекту, використовувані з початку створення застосування.

Хоча при створенні нового проекту в середовищі Visual Studio .NET пропонується досить великий список типів проектів, але насправді є всього три основні різновиди застосувань – **Windows Application, Console Application** і **Class Library**.

Все інше – це їх різні варіації за рахунок використання певних шаблонів або майстрів (саме тому праве вікно і називається **Templates**), які забезпечують автоматичне виконання якихось початкових дій, які за бажання можна виконати і «вручну» (зокрема змінивши і базовий тип застосування). Користувач може підключити і свої власні варіанти шаблонів.

При виконанні команди **File | New Project...** чи клацанні на посиланні **New Project...** на початковій сторінці відкривається діалогове вікно **New Project** (**рис. 2.3**).

За замовчуванням Visual Studio присвоює новому проекту і рішенню на базі шаблону **Windows Forms Application** ім'я **WindowsFormsApplication1** (див. **рис. 2.3**). Виберіть шаблон **Windows Forms Application** і клацніть на кнопці **OK**, щоб перевести IDE в режим конструювання (**рис. 2.3**). Функції цього режиму призначені для створення графічного інтерфейсу застосування.

2.3. Основні частини візуального середовища розробки Visual Studio

Існує три основні частини візуального середовища для розробки проекту. У центрі знаходиться головне вікно для створення візуальних форм і написання коду. У ньому розміщуються вікна: **Solution Explorer** (**Обозреватель решений**), *інспектор властивостей* – **Properties Explorer (Свойства)**. Їх розміщення можна налаштовувати за своїми вподобаннями. Про те як це робиться дещо пізніше. Вікно **Solution Explorer** дозволяє побачити, з

яких проектів складається рішення і які файли входять до складу цих проектів. Вікно властивостей (**Properties**) містить список атрибутів об'єкта, виділеного в даний момент. У середовищі розробки наявне вікно **Toolbox (Панель элементов)** (див. **рис. 2.4**).

Середовище розробки Visual Studio .NET містить два типи вікон – **вікна інструментів** і **вікна документів**. *Вікна інструментів* (частина з яких була описана вище) доступні з допомогою команд меню **View** (**Вид**) і деяких інших, і їх доступність залежить від типу застосування і від того, які модулі розширення (додаткові утиліти та інструменти, зокрема розроблені сторонніми розробниками) додані до середовища розробки. У *вікнах документів* можна редагувати компоненти проектів.

З вікнами інструментів можна здійснювати різні маніпуляції. Зокрема, можна змусити їх автоматично з'являтися і зникати, групувати їх у вигляді багатосторінкового блокнота, варіювати їх розташування в середовищі розробки, робити їх «плаваючими» і навіть відображати на додатковому моніторі, якщо використання такого підтримується ОС.

Деякі вікна інструментів, наприклад вікно **Web Browser**, можна створювати у вигляді декількох екземплярів (це можна зробити, вибравши пункт меню **Windows | New Window**).

Можна також змусити вікна інструментів автоматично зникати, якщо вони в даний момент не є активними – у цьому випадку на екрані відображаються назва і піктограма вікна, над якою можна помістити покажчик миші, якщо вікно потрібно відобразити цілком. Для закріплення вікна на екрані, треба клацнути мишею по зображенню канцелярської кнопки на заголовку цього вікна.

2.4. Форми та елементи управління

Прямокутник із заголовком **Form1** (*форма*) є головним вікном створюваного застосування Windows Forms. Застосування Visual C# можуть містити декілька **форм** (*вікон*). Visual Studio містить багато *готових елементів управління* та *інших компонентів*, які можуть використовуватися для створення і налаштування поведінки застосувань.

Тут ми використовуватимемо готові елементи управління з **.NET Framework Class Library**. Розмістивши елемент на формі, розробник може змінити його властивості. Наприклад, на **рис. 2.5**

показано, де змінюється текст заголовка форми, а на **рис. 2.6** зображене діалогове вікно для вибору властивостей шрифту в елементі управління.

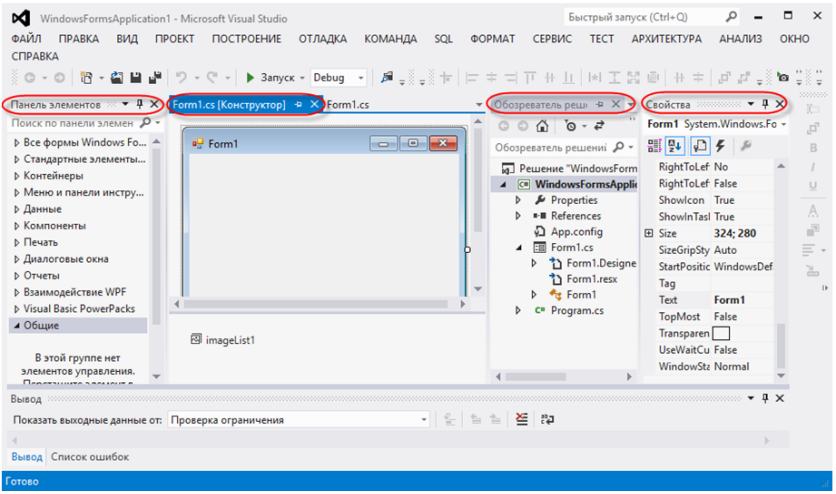


Рис. 2.4. Основні вікна візуального середовища розробки Visual Studio

Форма та елементи управління утворюють *графічний інтерфейс застосування*. Користувач набирає дані на клавіатурі, клацає кнопками миші або вводить їх іншими способами. Графічний інтерфейс використовується для відображення інструкцій та іншої інформації, призначеної для користувача. Наприклад, у діалоговому вікні **New Project** на **рис. 2.4** користувач клацає кнопкою миші, щоб вибрати тип шаблону, а потім вводить назву проекту з клавіатури.

Ім’я кожного відкритого документа виводиться на *корінці вкладки*. Щоб переглянути документ за наявності декількох відкритих документів, клацніть на відповідному корінці. Активна вкладка (вкладка з поточним документом/активна вкладка) виділена синім кольором (наприклад, **Form1.cs [Design]** на **рис. 2.7**).

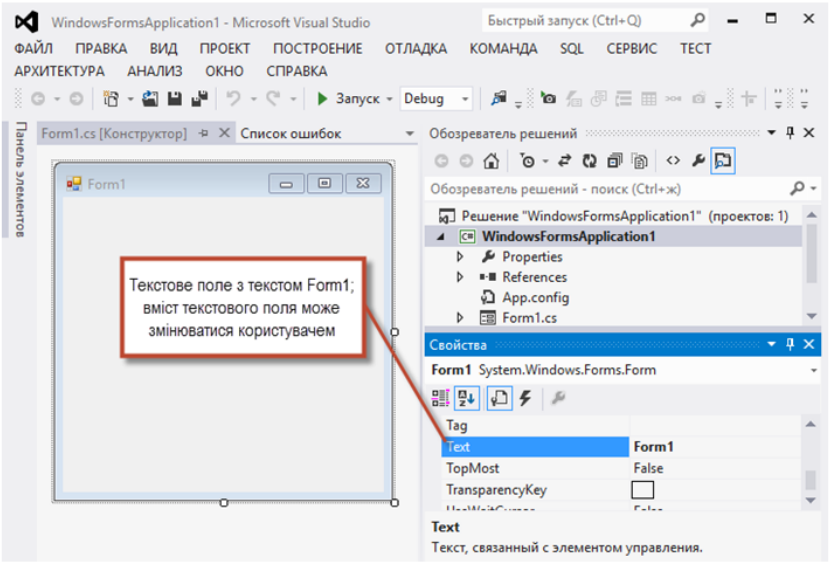


Рис. 2.5. Текстове поле для зміни властивості у Visual Studio IDE

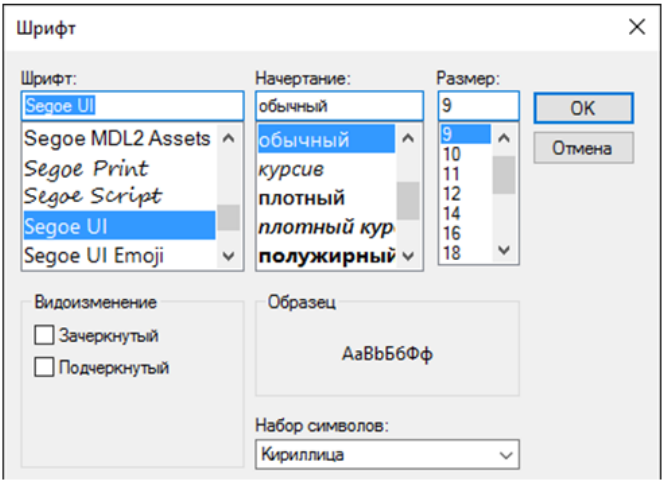


Рис. 2.6. Діалогове вікно для вибору властивостей шрифту елемента управління

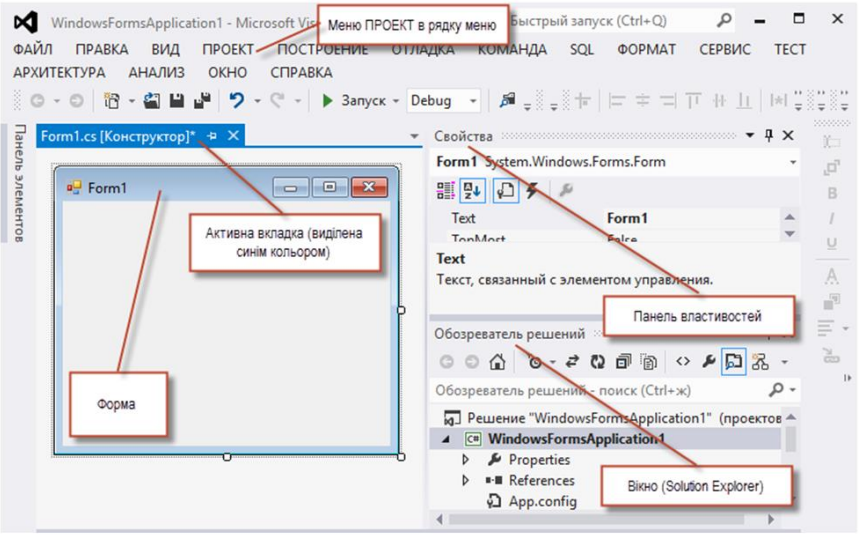


Рис. 2.7. Режим конструювання (**Design view**) в IDE

2.5. Рядок меню і панель інструментів

Основні команди управління розробкою, супроводом і виконанням застосувань об'єднані в ***меню***, які розміщуються в рядку меню IDE. Набір відображуваних меню, залежить від того, яка *операція зараз виконується* в IDE.

Меню містять групи взаємозв'язаних команд, при виборі яких IDE виконує конкретні дії – наприклад, відкриває вікно, зберігає файл, виводить файл на друк або виконує застосування. Наприклад, нові проекти створюються командою **File | New Project...**. У **таблиці 2.1** приведений короткий опис меню.

Таблиця 2.1. Меню Visual Studio, яке відображаються в режимі конструювання

Меню	**Опис**
File (Файл)	Команди відкриття, закриття, додавання та збереження проектів, а також друк цих проектів і виходу з Visual Studio.
Edit (Правка)	Команди редагування застосувань: копіювання, вирізування, вставки, відміни, повтору,

	видалення, пошуку і виділення.
View (Вид)	Команди відображення вікон IDE (наприклад, Solution Explorer, панелі інструментів і вікна властивостей) і додавання панелей інструментів в IDE.
Project (Проект)	Команди управління проектами та їх файлами.
Build (Построение)	Команди перетворення застосування у виконувану програму.
Debug (Отладка)	Команди компіляції, налаштування (тобто виявлення і виправлення помилок) і запуску застосувань.
Team (Команда)	Під'єднання до сервера Team Foundation Server для груп розробки, в яких над одним застосуванням працюють відразу декілька учасників.
Format (Формат)	Команди, які управляють розміщенням і характеристиками елементів управління форми. Меню FORMAT виводиться ТІЛЬКИ при виділенні управління у режимі конструювання.
Tools (Сервис)	Команди виклику додаткових інструментів і засобів налаштування IDE.
Test (Тест)	Команди для виконання різних видів автоматизованого тестування в застосуваннях.
Window (Окно)	Команди для згортання, відкриття, закриття і відображення вікон IDE.
Help (Справка)	Команди для звернення до довідкових засобів IDE.

Багато часто використовуваних команд меню можуть виконуватися з панелі інструментів (**рис. 2.8**). Кнопки, які знаходяться на панелі, є графічними представленнями команд. За замовчуванням при першому запуску Visual Studio відображається стандартна панель інструментів з кнопками часто використовуваних команд: **відкриття файлів, додавання нових елементів в проект, збереження файлів, запуску застосувань та ін.** (див.**рис. 2.8**). Склад кнопок на стандартній панелі інструментів залежить від версії Visual

Studio. Деякі команди спочатку заблоковані (недоступні для використання) і стають доступними тільки при потребі. Наприклад, команда збереження файла стає доступною тоді, коли ви починаєте редагувати файл.

Щоб змінити набір відображуваних панелей інструментів, треба виконати команду **View | Toolbars (Вид | Панели инструментов)** і вибрати потрібну панель у списку (**рис. 2.9**). Кожна вибрана панель інструментів відображається разом з іншими панелями у верхній частині вікна Visual Studio. Щоб перемістити панель інструментів, відмітьте її у лівій частині панелі. Щоб виконати команду з панелі інструментів, клацніть на її піктограмі.

Якщо вам складно запам'ятати, яку команду представляє та або інша кнопка на панелі інструментів, то наведіть покажчик миші на кнопку і затримайте його, після невеликої паузи з'являється підказка з описом (**рис. 2.10**). Підказки допомагають опанувати можливості середовища розробки IDE і нагадують, що робить кожна з кнопок панелей.

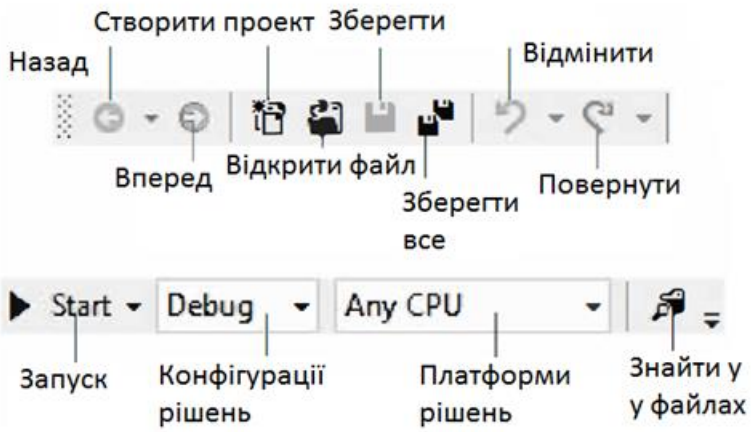


Рис. 2.8. Стандартна панель інструментів Visual Studio

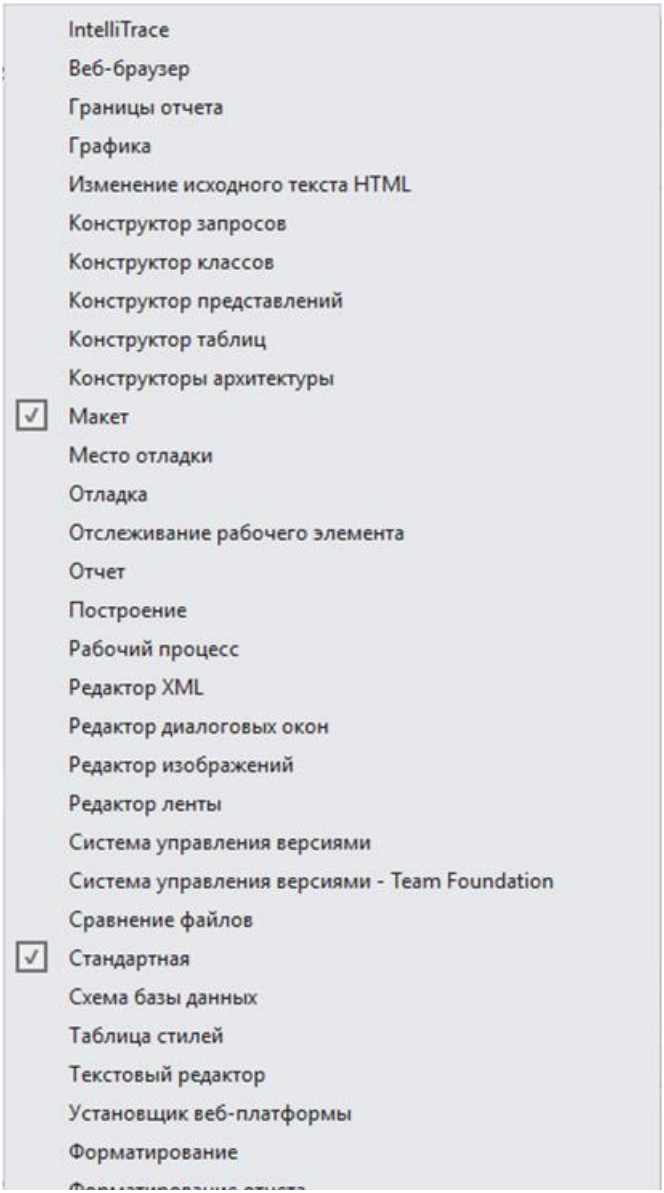


Рис. 2.9. Список панелей інструментів, які можна додати в IDE

2.6. Переміщення у Visual Studio IDE

IDE надає вікна для роботи з файлами проекту і налаштування елементів управління. Розглянемо деякі вікна, часто використовувані при розробці застосувань Visual C#. Щоб звернутися до будь-якого з цих вікон, виберіть його ім'я в меню **View (Вид)**.

Для економії місця у Visual Studio реалізована функція автоматично приховуваних вікон. Коли для вікна включений цей режим, то біля лівого, правого або нижнього краю вікна IDE з'являється корінець з ім'ям вікна (**рис. 2.11**). Якщо клацнути на ньому, вікно з'являється на екрані (**рис. 2.12**). Повторне клацання на імені вікна (або за його межами) приховує його. Щоб «закріпити» вікно (тобто відключити режим автоматично приховуваних вікон і залишити вікно відкритим), клацніть на кнопці із *зображенням шпильки*. При включеному режимі автоматично приховуваних вікон шпилька на кнопці розташована *горизонтально* (**рис. 2.12**), а коли вікно «закріплене», шпилька розміщена *вертикально* (**рис. 2.13**).

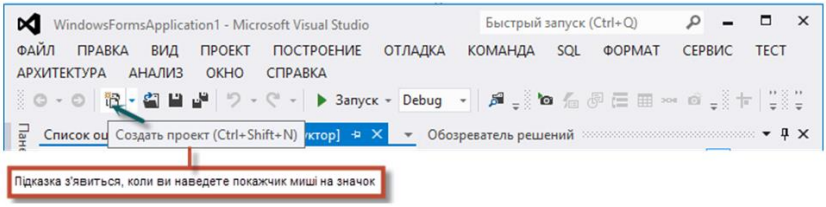


Рис. 2.10. Демонстрація підказки

Далі ми розглядатимемо три головні вікна Visual Studio – *оглядач рішень* **Solution Explorer**, *вікно властивостей* **Properties** і *вікно панелі елементів* **Toolbox**. Ці вікна виводять інформацію про проект і допомагають будувати застосування.

2.6.1. Solution Explorer

Вікно **Solution Explorer** (**рис. 2.14**) надає доступ до усіх файлів застосування. Якщо воно не відображається в IDE, виконайте команду **View | Solution Explorer**. Коли ви відкриваєте нове або існуюче рішення, його вміст виводиться у вікні **Solution Explorer**.

Стартовим проектом називається проект, який запускається при виборі команди **Debug | Start Debugging** (або натисненні клавіші

F5). Для рішень з одним проектом стартовий проект єдиний (у цьому випадку **WindowsFormsApplication1**). Ім'я стартового проекту виділяється жирним шрифтом у вікні **Solution Explorer**. При першому створенні застосування вікно **Solution Explorer** виглядає так, як показано на **рис. 2.15**. Файл Visual C#, який відповідає формі на **рис. 2.4** і **2.7**, називається **Form1.cs** (виділений на **рис. 2.15**). Файли Visual C# використовують розширення **.cs**.

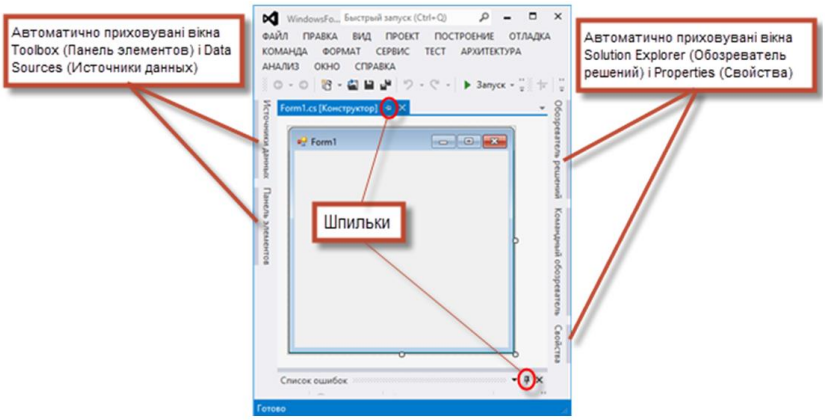


Рис. 2.11. Режим автоматично приховуваних вікон

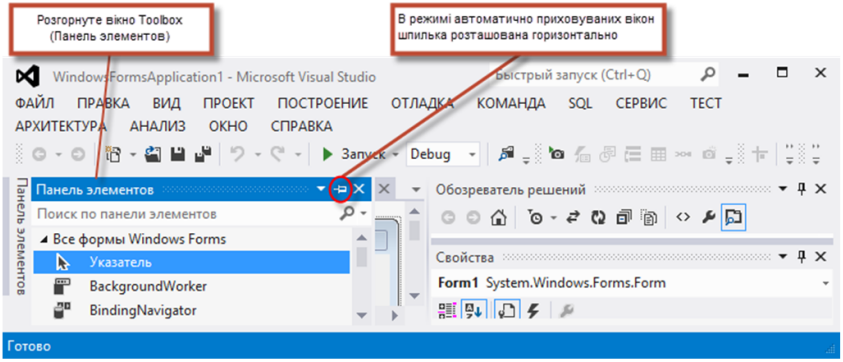


Рис. 2.12. Відображення прихованого вікна **Toolbox**

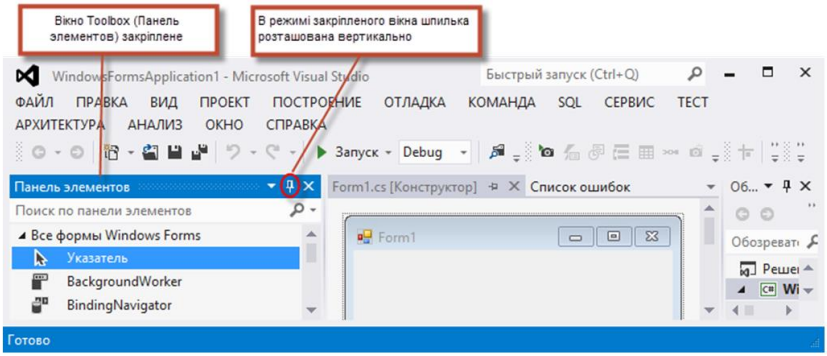


Рис. 2.13. Режим автоматично приховуваних вікон відключений – вікно «закріплене»

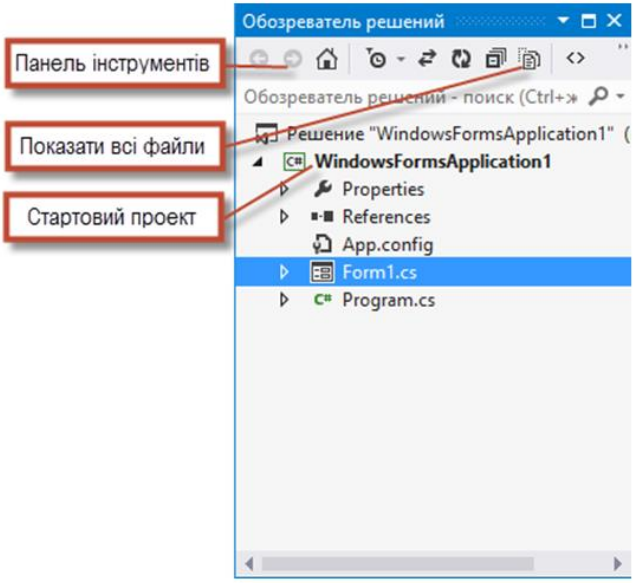


Рис. 2.14. Вікно **Solution Explorer** з відкритим проектом

За замовчуванням в IDE відображаються тільки ті файли, які вам, можливо, доведеться редагувати; інші файли, згенеровані IDE,

залишаються прихованими. У вікні **Solution Explorer** присутня панель інструментів з декількома кнопками. Кнопка **Show All Files (Показать все файлы)** (див. **рис. 2.14**) відображає всі файли рішення, включаючи згенеровані IDE. Клацання на стрілці зліва від вузла розгортає або згортає цей вузол. Спробуйте клацнути на стрілці зліва від **References**, щоб переглянути елементи, згруповані під цим заголовком (**рис. 2.15**). Згорніть дерево повторним клацанням на стрілці. Ця схема також використовується в інших вікнах Visual Studio.

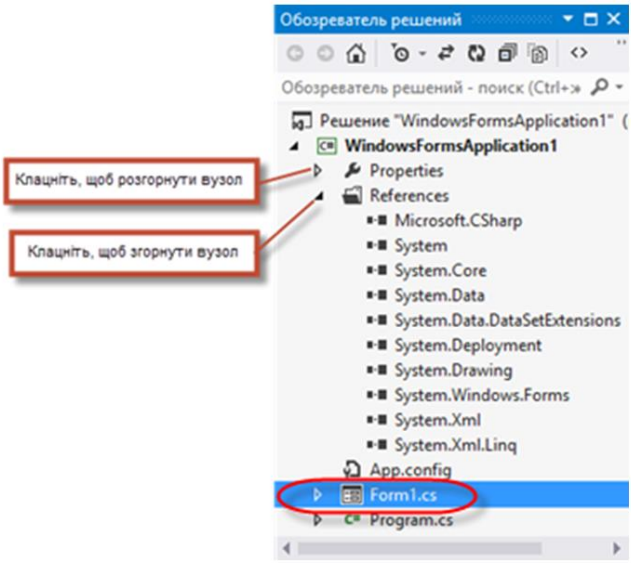


Рис. 2.15. Вікно **Solution Explorer** з розгорнутим вузлом **References**

2.6.2. Панель елементів

Щоб викликати *вікно панелі елементів* (**Toolbox**), виконайте команду **View | Toolbox**. У ньому містяться елементи, які використовуються для модифікації форм (**рис. 2.16**). У методології візуального програмування розробник «перетягує» елементи управління на форму, а IDE генерує код створення цих елементів. Так код створюється простіше і швидше, ніж при самостійному написанні. Щоб керувати машиною, не обов'язково знати, як

влаштований її двигун; так само не обов'язково уміти будувати елементи, щоб використати їх. Повторне використання готових елементів економить час і гроші при розробці. Ми використаємо панель елементів пізніше, при створенні нашого першого застосування.

Готові елементи управління на панелі елементів згруповані за категоріями (див. **рис. 2.16**). І знову зверніть увагу на стрілки, використовувані для згортання або розгортання груп.

2.6.3. Вікно властивостей

Якщо вікно властивостей (**Properties**) не відображається під вікном **Solution Explorer**, виконайте команду **View | Properties Window (Вид | Окно свойств)**. У вікні властивостей виводяться властивості поточної виділеної форми, елемента управління або файла. Властивості описують характеристики форми або елемента управління: *розмір, колір, позицію* і т. д. Різні форми та елементи управління мають різні набори властивостей – опис яких виводиться в нижній частині вікна властивостей при їх виділенні.

На **рис. 2.17** зображено вікно властивостей форми **Form1**. У лівому стовпці перераховані властивості форми, а в правому виводяться поточні значення кожної властивості. Список властивостей можна відсортувати або за абеткою (кнопка **Alphabetical**), або за категоріями (кнопка **Categorized**). Залежно від розміру вікна властивостей деякі властивості можуть не поміститися на екрані. Щоб прокрутити список властивостей, використайте смугу прокручування або кнопки зі стрілками біля верхнього і нижнього краю смуги. Про те, як задавати значення властивостей, буде розказано пізніше.

Вікно властивостей грає найважливішу роль у візуальній розробці – воно дозволяє змінювати властивості елементів управління на візуальному рівні, без написання коду. Розробник бачить, які властивості можна змінити, а у багатьох випадках бачить діапазон допустимих значень заданої властивості. У вікні властивостей виводиться короткий опис виділеної властивості, що допомагає зрозуміти його призначення. Властивість можна швидко задати у цьому вікні, без написання будь-якого коду.

У верхній частині вікна властивостей розташований розкривний список, для вибору компонентів. У ньому можна вибрати форму або елемент управління, властивості яких ви хочете вивести у вікні властивостей, без виділення цієї форми/елемента управління у графічному інтерфейсі.

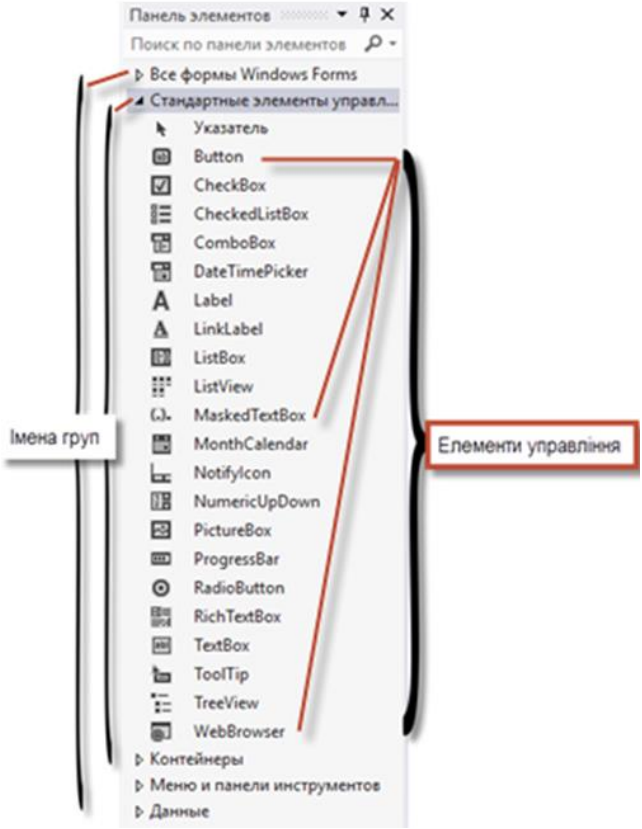


Рис. 2.16. Вікно **Toolbox** з елементами управління з групи **Common Controls**

2.7. Довідкова система

Компанія Microsoft надає велику довідкову документацію в *меню* **Help** – це відмінний спосіб швидко отримати потрібну інформацію про Visual Studio, Visual C# і багато чого іншого.

2.7.1. Контекстна підказка

Visual Studio надає *контекстну довідку* щодо «поточного» об'єкта, який знаходиться під покажчиком миші. Щоб отримати контекстну довідку, клацніть на об'єкті і натисніть клавішу **F1**. Довідкова документація відображається у вікні браузера. Щоб повернутися в IDE, або закрийте вікно браузера, або виберіть кнопку IDE на панелі завдань Windows. На **рис. 2.18** зображена довідкова сторінка для властивості **Text** об'єкта форми. Щоб переглянути її, виділіть форму, клацніть на її властивості **Text** у вікні властивостей і натисніть клавішу **F1**.

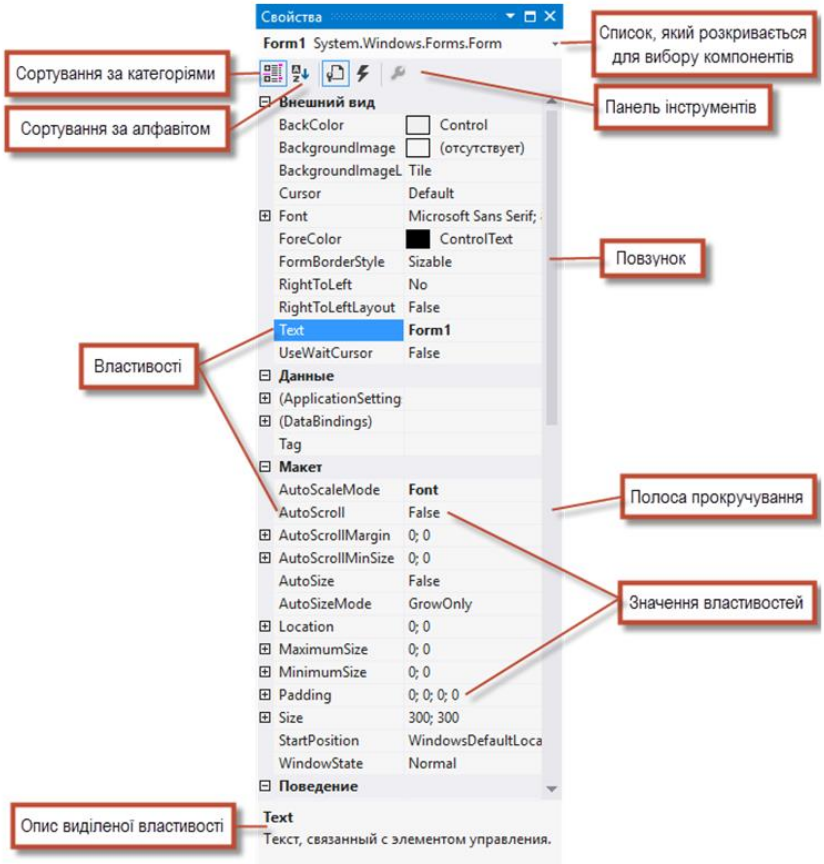


Рис. 2.17. Вікно властивостей

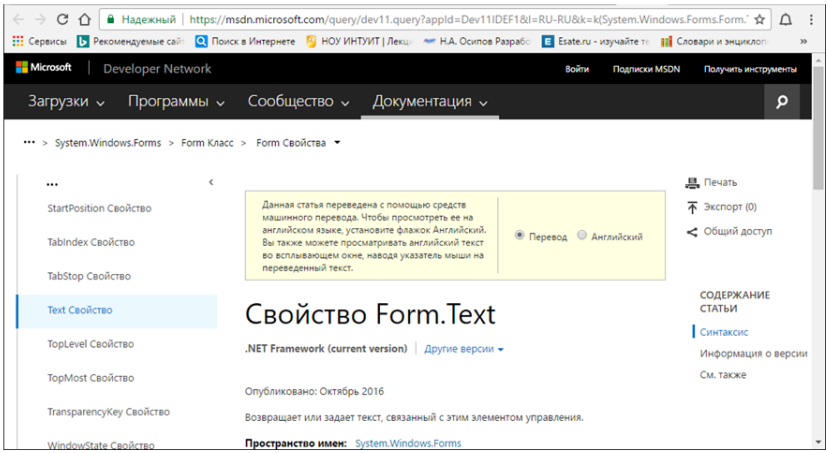


Рис. 2.18. Контекстна довідка

2.8. Нескладне застосування з текстом і графікою

Зараз ми створимо застосування, яке виводить вітальне повідомлення і комічне зображення. Застосування складається з форми з двома елементами управління: *написом* (**Label**) і *графічним полем* (**PictureBox**). Вікно виконуваного застосування зображене на **рис. 2.19**.

При створенні цього застосування *ми не напишемо ні єдиного рядка коду*. Замість цього використовуватиметься методологія візуальної розробки застосувань: Visual Studio обробляє ваші дії (клацання кнопкою миші, перетягання) і генерує код застосування.

Візуальне програмування добре підходить для графічних застосувань, які передбачають істотну взаємодію з користувачем. Щоб створити, зберегти, запустити і завершити своє перше застосування, виконайте наступні дії:

1. *Створіть новий проект.* Щоб створити нове застосування **Windows Forms**, виконайте команду **File | New Project...** для виклику діалогового вікна **New Project** (**рис. 2.20**). Виберіть шаблон **Windows Forms Application**. Введіть ім'я проекту **Нескладне застосування**, вкажіть теку для

збереження у рядку **Расположение** і клацніть на кнопці **ОК**. Як було показано раніше, при створенні нового застосування **Windows Forms** середовище розробки відкривається у *режимі конструювання* (тобто застосування знаходиться в процесі побудови, а не виконання).

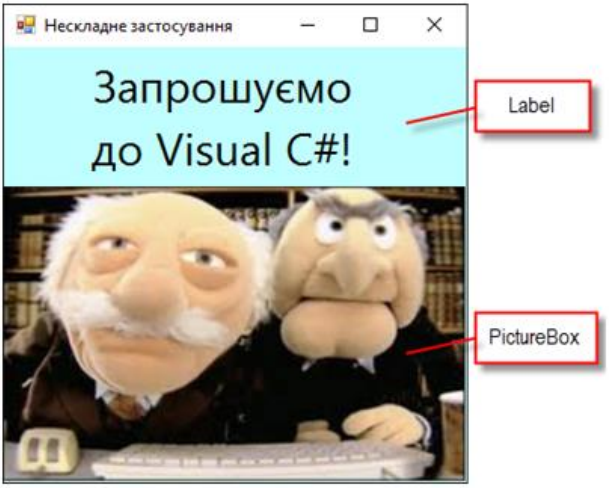


Рис. 2.19. Просте застосування

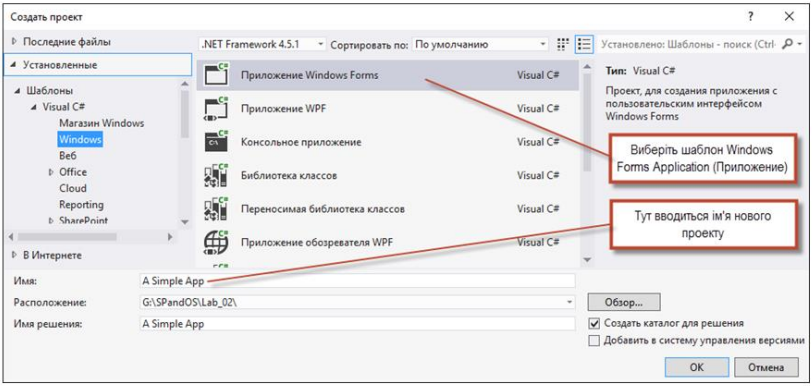


Рис. 2.20. Діалогове вікно **New Project**

2. *Введіть текст рядка заголовка форми.* Цей текст визначається властивістю **Text** об'єкта **Form** (**рис. 2.21**). Якщо вікно властивостей не відкрите, виконайте команду **View | Properties Window (Вид | Окно свойств)**. Клацніть у будь-якій точці форми, щоб викликати набір властивостей форми у вікні властивостей. У текстовому полі праворуч від властивості **Text** введіть рядок **Нескладне застосування**, як показано на **рис. 2.21**. Натисніть клавішу **Enter**; заголовок форми негайно оновиться (**рис. 2.22**).

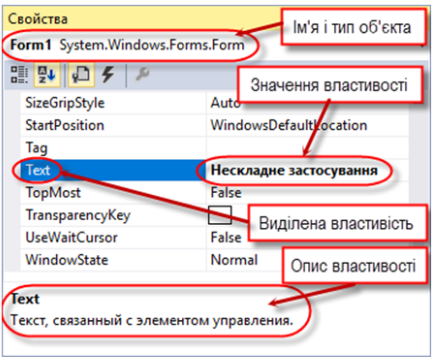


Рис. 2.21. Задання властивості **Text** у вікні властивостей

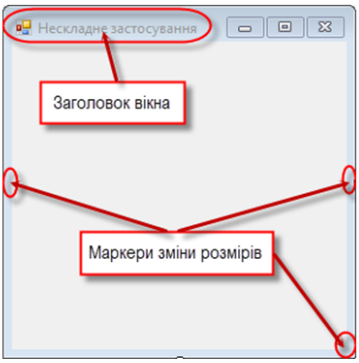


Рис. 2.22. Форма з маркерами зміни розмірів

3. *Змініть розміри форми.* Перетягніть правий нижній маркер зміни розмірів (маленькі білі квадратики навколо форми на **рис. 2.22**), щоб збільшити розміри форми (**рис. 2.23**).
4. *Змініть колір фону форми.* Властивість **BackColor** задає колір фону форми або елемента управління. Якщо клацнути на властивості **BackColor** у вікні властивостей, поруч зі значенням властивості з'являється стрілка (**рис. 2.24**). При клацанні на кнопці зі стрілкою виводиться набір можливих значень, залежний від властивості. У цьому прикладі виводиться набір вкладок **Custom, Web** і **System** (за замовчуванням). Клацніть на кнопці **Custom**, щоб викликати палітру (сітку із зразками кольорів). Клацніть на квадратику *світло-синього* кольору. При виборі кольору палітра закривається, а фон форми забарвлюється у світло-синій колір (**рис. 2.25**).

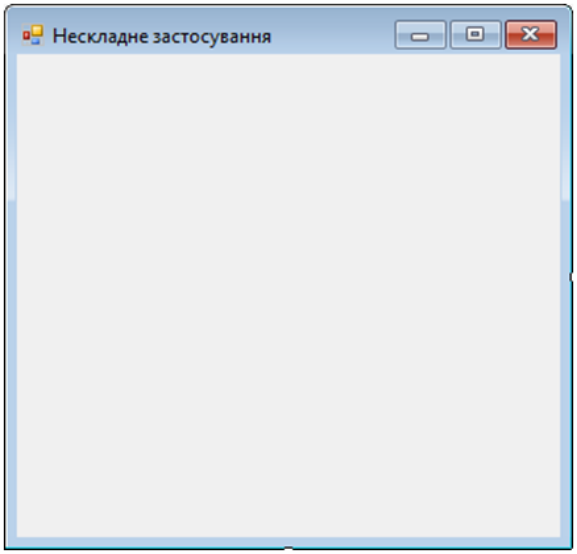


Рис. 2.23. Змінена форма

5. *Додайте на форму напис.* Якщо вікно панелі елементів не відображається, виконайте команду **View | Toolbox**. Для створюваного тут типу застосування, використовуються елементи управління з групи **All Windows Forms** або **Common Controls**. Якщо якась з цих груп згорнута, розгорніть її клацанням на стрілці зліва від імені групи (**рис. 2.16**). Потім двічі клацніть на елементі управління **Label** у вікні панелі елементів. У лівому верхньому кутку форми з'являється напис (див. **рис. 2.26**).

   І хоча подвійне клацання на будь-якому елементі управління у вікні панелі елементів розміщує його на формі, ви також можете «перетягувати» елементи управління з панелі елементів на форму – і можливо, цей спосіб вам здається зручнішим, тому що елемент можна розмістити у будь-якому потрібному місці. У написах за замовчуванням відображається текст **label1**. При додаванні напису на форму IDE задає значення властивості **BackColor** напису

властивості **BackColor** форми. Щоб змінити колір фону напису, змініть її властивість **BackColor**.

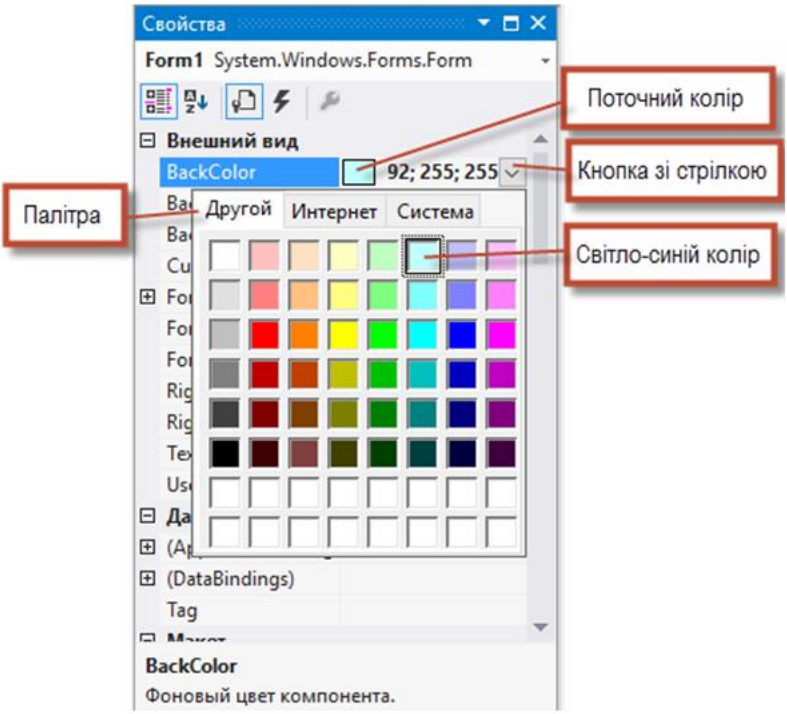


Рис. 2.24. Зміна властивості **BackColor** форми

1. *Змініть параметри оформлення напису*. Клацніть на написі, щоб виділити його. Властивості елемента управління відображаються у вікні властивостей. Властивість **Text** елемента управління **Label** визначає поточний текст напису. У кожного з елементів управління **Form** і **Label** є власна властивість **Text** – форми та елементи управління можуть мати однойменні властивості (**BackColor**, **Text** і т. д.), це не викликає конфліктів. Задайте властивості **Text** напису значення «**Запрошуємо до Visual C#!**». Розміри елемента управління автоматично змінюються так, щоб увесь текст помістився в одному рядку. Щоб задати розміри елемента

управління **Label** вручну, задайте його властивості **AutoSize** значення **False**. Змініть розміри напису (використовуючи маркери зміни розмірів), щоб текст поміщався в елементі. Розмістіть напис біля верхнього краю форми по центру; використайте перетягання мишею або клавіші ← і → (**рис. 2.27**). Для центрування напису можна виділити його і виконати команду **Format | CenterInForm | Horizontally (Формат | По центру формы | По горизонтали)**.

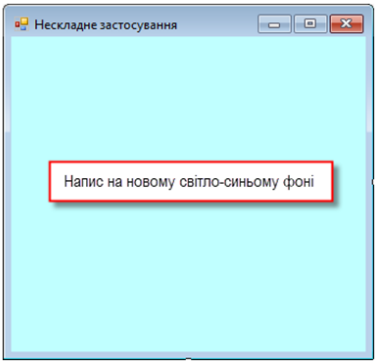


Рис. 2.25. Форма зі зміненою властивістю **BackColor**

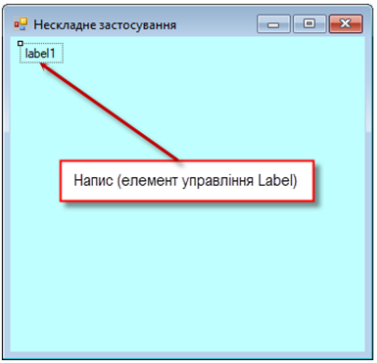


Рис. 2.26. Додавання напису на форму

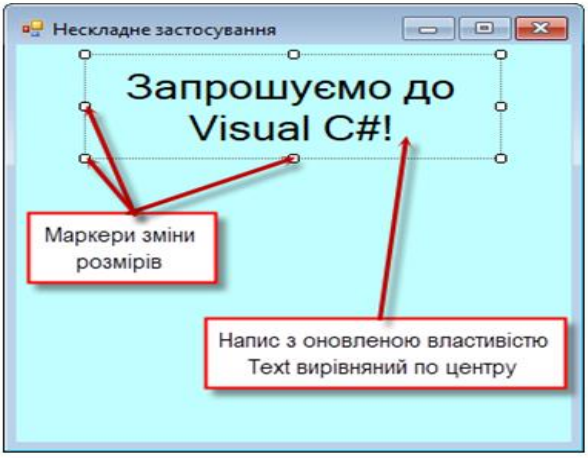


Рис. 2.27. Форма після зміни властивостей **Form** і **Label**

2. *Задайте розмір шрифту напису.* Щоб змінити гарнітуру і зображення тексту, виділіть значення властивості **Font**; поряд зі значенням з'являється кнопка з багатьма крапками (**рис. 2.28**). Якщо клацнути на ній, на екрані з'являється діалогове вікно для вибору додаткових значень – тут це вікно **Font** (**рис. 2.29**). У цьому вікні можна вибрати ім'я шрифту (список доступних шрифтів залежить від системи), зображення (звичайне, курсивне, напівжирне і т. д.) і розмір шрифту (**16, 18, 20** і т. д.). У полі **Sample** виводиться зразок тексту, оформленого заданим шрифтом.

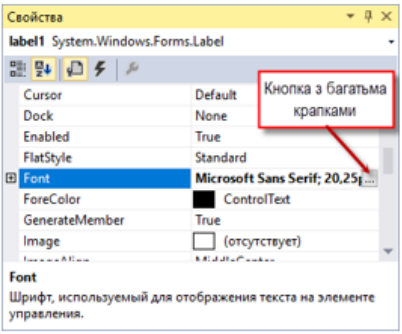


Рис. 2.28. Вікно властивостей з властивістю напису **Font**

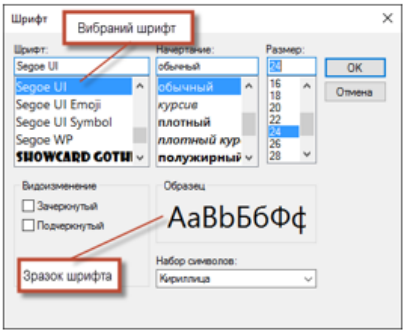


Рис. 2.29. Діалогове вікно **Font** для вибору шрифта, зображення і розміру

Виберіть для властивості **Font** значення **Segoe UI** – шрифт, рекомендований Microsoft для інтерфейсів користувача. Задайте властивості **Size** значення **24** і клацніть на кнопці **ОК**. Якщо текст напису не поміщається в одному рядку, він переноситься на наступний рядок. Змініть розміри напису так, щоб слова «**Запроушємо до**» виводилися в першому рядку, а слова «**Visual C#!**» – в другому. Знову вирівняйте напис по центру форми.

3. *Вирівняйте текст напису.* Задайте властивість **TextAlign** напису, яка визначає спосіб вирівнювання тексту. На екрані з'являється сітка **3×3** з кнопок, які представляють різні способи вирівнювання (**рис. 2.30**). Позиція кожної кнопки відповідає місцезнаходженню тексту в написі. Для нашого застосування задайте властивості **TextAlign** значення

MiddleCenter – в цьому варіанті текст напису вирівнюється по центру по горизонталі і вертикалі (див. **рис. 2.27**). Інші значення **TextAlign** дозволяють розташувати текст в потрібному місці напису. З деякими способами вирівнювання вам доведеться змінити розміри напису, щоб текст краще розміщувався у ньому.

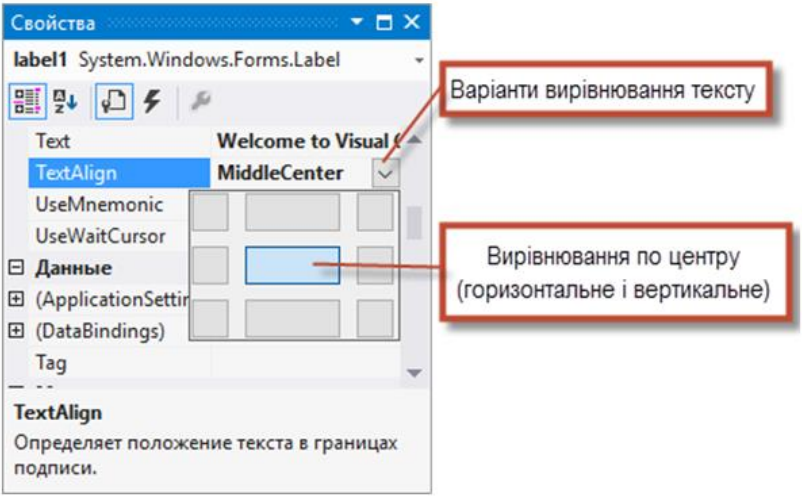


Рис. 2.30. Вирівнювання тексту по центру напису

4. *Додайте на форму графічне поле – елемент управління для виведення графіки.* Це робиться приблизно так само, як на кроці 3, коли ми додали на форму напис. Знайдіть на панелі елементів (див. **рис. 2.16**) елемент **PictureBox** і здійсніть на ньому подвійне клацання. Коли графічне поле з'явиться на формі, перемістіть його в нижню частину форми – використайте перетягання мишею або клавіші зі стрілками (**рис. 2.31**).

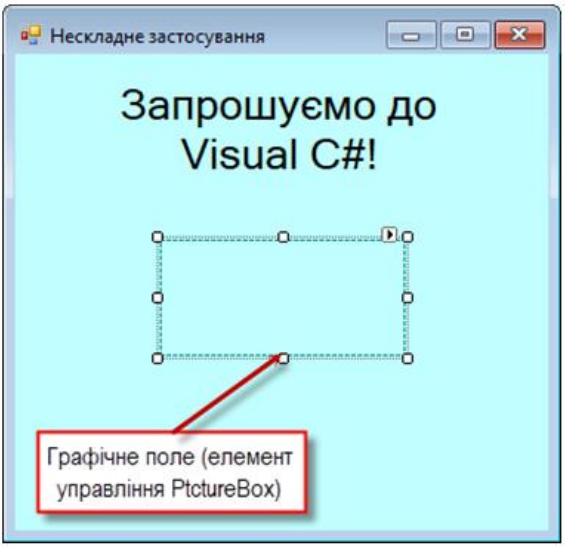


Рис. 2.31. Вставка і вирівнювання графічного поля

5. *Вставте зображення.* Клацніть по кнопці на графічному полі, щоб вивести його властивості у вікні властивостей (**рис. 2.32**). Знайдіть і виділіть властивість **Image**; через те, що зображення не вибране, з'являється рядок **none (отсутствует)**. Клацніть на кнопці з багатьма крапками (або посилання **Choose Image...** над описом властивості), щоб викликати діалогове вікно **Select Resource (Выбор ресурса)** (**рис. 2.33**), призначене для імпортування файлів, використовуваних застосуванням, – наприклад, графіки. Клацніть на кнопці **Import...**, знайдіть теку із зображенням, виділіть графічний файл і клацніть на кнопці **ОК**. Ми використали файл заготовлений заздалегідь. Ескіз зображення з'являється в діалоговому вікні **SelectResource** (**рис. 2.34**). Клацніть на кнопці **OK**, щоб використати зображення. Підтримуються такі графічні формати, як **PNG (Portable NetworkGraphics), GIF (Graphic Interchange Format), JPEG (Joint Photographic ExpertsGroup)** і **BMP (Windows Bitmap)**. Щоб масштабувати зображення за розмірами **PictureBox**, задайте властивості **SizeMode**

значення **Stretchimage** (**рис. 2.35**). Збільшіть розміри графічного поля (**рис. 2.36**).

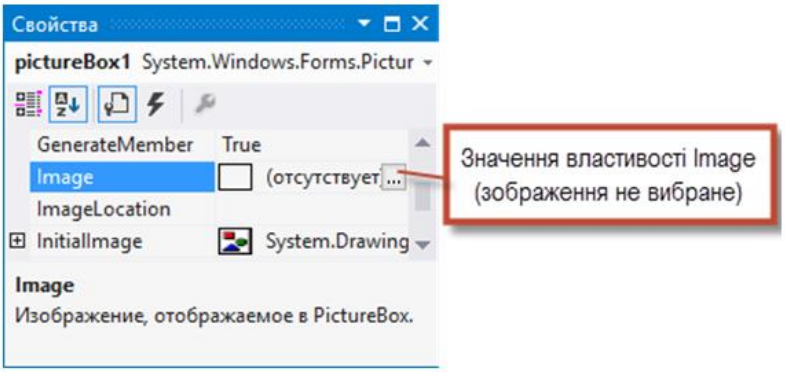


Рис. 2.32. Властивість **Image** елемента управління **PictureBox**

6. *Збережіть проект.* Виконайте команду **File | Save All**, щоб зберегти все рішення. У файлі рішення (розширення **.sln**) зберігається ім'я і місцезнаходження проекту, а файл проекту (розширення **.csproject**) містить імена і місцезнаходження всіх файлів проекту. Якщо потрібно знову відкрити свій проект в майбутньому, просто відкрийте його файл **.sln**.

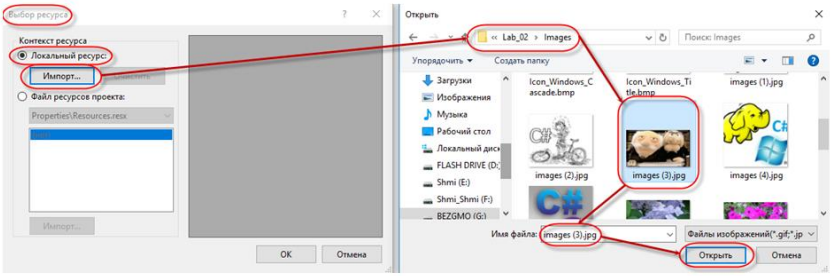


Рис. 2.33. Діалогове вікно **Select Resource** для вибору зображення

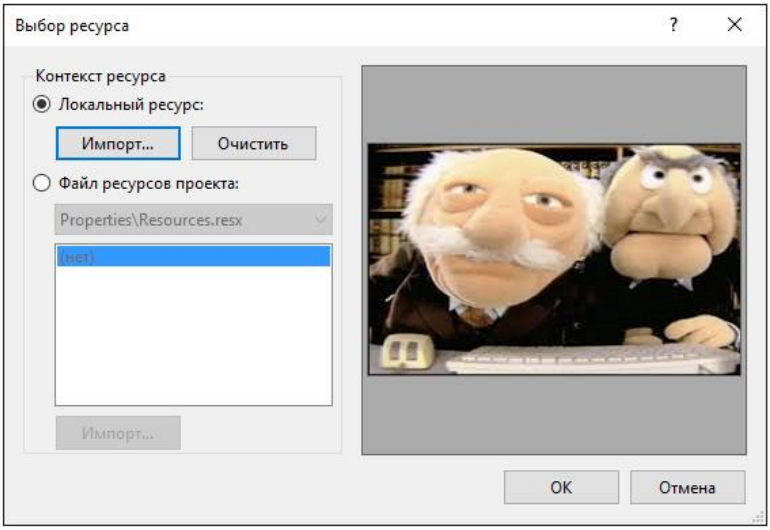


Рис. 2.34. Діалогове вікно **Select Resource** з вибраним зображенням

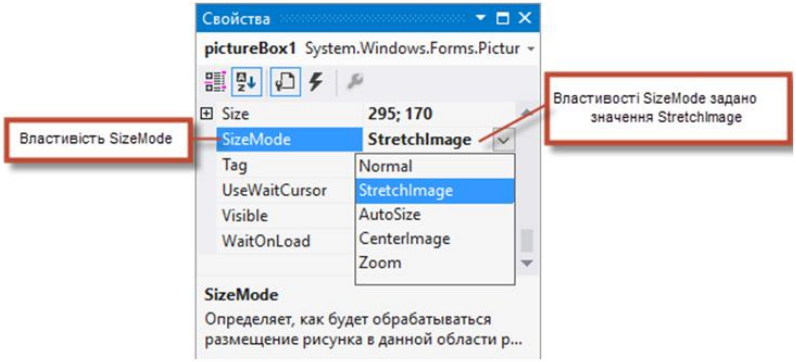


Рис. 2.35. Масштабування зображення за розмірами **PictureBox**

Рис. 2.36. Графічне поле з вибраним зображенням

7. *Запустіть проект.* Згадайте, що до цього моменту ми працювали в режимі конструювання IDE (тобто застосування створювалося, а не виконувалося). У режимі виконання багато функцій IDE недоступні – в інтерфейсі такі функції виділяються сірим кольором. Текст **Form1.cs [Design]** на вкладці проекту (**рис. 2.37**) означає, що форма будується візуальним, а не програмним способом. Якби ми писали програмний код, то вкладка містила б тільки текст **Form1.cs**. Якщо текст на вкладці закінчується зірочкою **(*)**, значить, файл містить не збережені зміни. Виконайте команду **Debug | Start Debugging** (або натисніть клавішу **F5**), щоб запустити застосування. На **рис. 2.38** зображене середовище IDE в режимі виконання. Багато кнопок панелей інструментів і меню під час виконання стають недоступними. Виконувані застосування з'являються в

окремому вікні за межами IDE, як показано в правій нижній частині **рис. 2.38**.

8. *Завершіть застосування.* Клацніть на кнопці закриття (в правому верхньому кутку виконуваного застосування). Ця дія припиняє виконання застосування і повертає IDE в режим проектування. Також застосування можна завершити командою **Debug | Stop Debugging**.

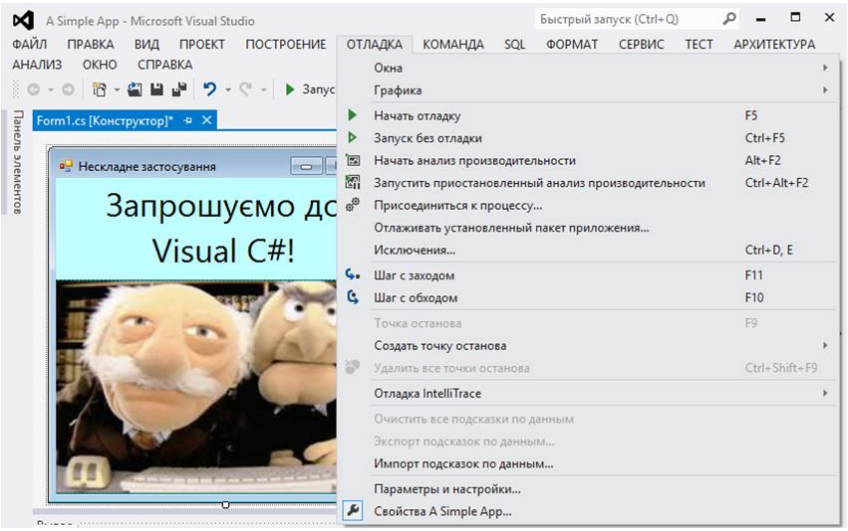


Рис. 2.37. Налаштування рішення

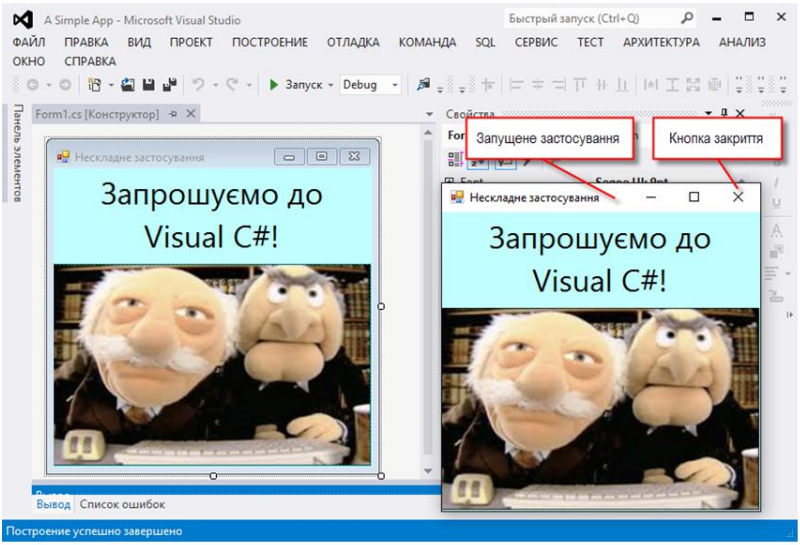


Рис. 2.38. IDE в режимі виконання із запущеним застосуванням на передньому плані

2.9. Підсумки

У цьому розділі ми розглянули ключові можливості Visual Studio IDE. Ми створили працездатне застосування Visual C#, не написавши ні єдиного рядка коду. Розробка застосувань Visual C# використовує поєднання двох стилів: *візуальна розробка* дозволяє легко створювати графічні інтерфейси без написання рутинного коду, а *«традиційне»* програмування визначає *поведінку застосування*.

РОЗДІЛ 3. НАЛАШТУВАННЯ ЗАСТОСУВАНЬ

У середовищі Visual Studio передбачений потужний набір засобів побудови і налаштування. Завдяки конфігураціям побудови можна вибирати компоненти для побудови, виключати компоненти, а також визначати, як будуть побудовані вибрані проекти і для якої платформи.

Зробимо короткий огляд можливостей налаштування застосувань, які пропонуються середовищем Visual Studio, з концентрацією уваги на тих аспектах, які можуть виявитися новими для деяких категорій розробників.

3.1. Основні типи помилок

У процесі написання програмного коду можлива поява трьох видів помилок:

- **синтаксичні помилки** (syntax errors) виникають, якщо компілятор не в змозі зрозуміти переданий йому початковий текст;
- **логічні помилки** (logical errors) не перешкоджають компіляції і виконанню програми, але приводять до несподіваних результатів;
- **помилки періоду виконання** (run-time errors) виникають при спробі виконати неприпустиму дію під час виконання програми.

Синтаксичні помилки можна виявити відразу. Код програми за наявності синтаксичних помилок не буде скомпільований. Після завершення побудови застосування можна використовувати відлагоджувач для виявлення та усунення таких проблем, як логічні і семантичні помилки, виявлені під час виконання. Розглянемо процес виявлення та усунення цих помилок в середовищі Visual Studio детальніше.

3.1.1. Синтаксичні помилки

Синтаксичні помилки виникають, якщо компілятор не може скомпілювати наданий йому код, наприклад, через помилки в ключовому слові, пропуску розділового знака або використання

неправильної конструкції. Будь-яка з цих помилок не дозволить компілятору інтерпретувати такий код.

Середовище розробки Visual Studio полегшує пошук синтаксичних помилок, автоматично виявляючи їх при компонуванні проекту і виділяючи в початковому тексті підсвічуванням. Список виявлених помилок також виводиться у вікні **Error List (Список ошибок)** (**рис. 3.1**).

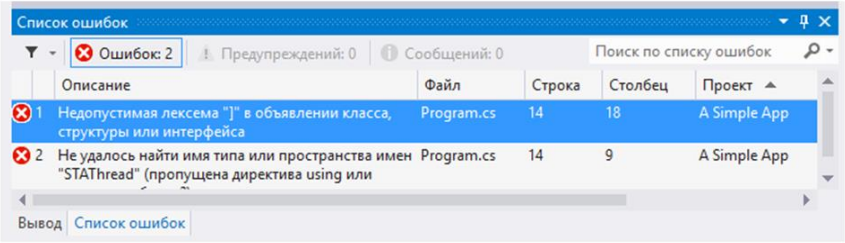


Рис. 3.1. Вікно **Error List**

Якщо двічі клацнути лівою кнопкою миші опис помилки у вікні **Error List**, середовище розробки перемістить курсор у відповідний рядок у вікні коду і виділить помилку підсвічуванням – зазвичай цього вистачає, щоб її розпізнати і виправити (див. **рис. 3.2**). Програмістові залишається тільки виправити знайдену помилку. Якщо для виправлення помилки вимагається додаткова інформація, можна скористатися вбудованою або онлайновою документацією MSDN, або іншими ресурсами співтовариства розробників.

3.1.2. Логічні помилки

Логічну помилку констатують, якщо застосування коректно компілюється і виконується, але ми не отримуємо очікувані результати. Діагностувати такі помилки важче, ніж інші, через відсутність вказівок на джерело помилки. Причиною логічної помилки можуть бути такі, на перший погляд, незначні промахи, як неправильно поставлена крапка в десятковому дробі або зайва ітерація в операторі циклу.

Щоб *виявити логічну помилку*, потрібно розробити тестові дані і дати можливість застосуванню на них виконатися, а потім проаналізувати результати виконання програми. Часто для

знаходження логічної помилки доводиться аналізувати рядок за рядком написаний код. Для цього застосовується режим покрокового виконання коду, який буде розглянутий далі.

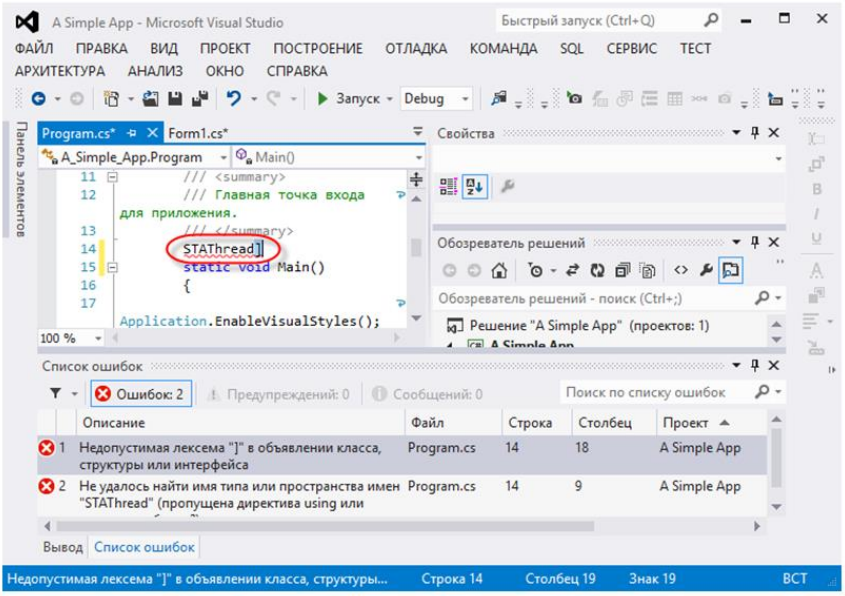


Рис. 3.2. Курсор у відповідному рядку у вікні коду, виділений підсвічуванням синього кольору

3.1.3. Помилки періоду виконання

Помилки періоду виконання виникають при спробі застосування *виконати неприпустиму операцію*. До цієї категорії відносяться ділення на нуль і дії, заборонені політиками безпеки: спроба їх виконання приводить до *генерації виключення захисту*.

При виникненні помилки періоду виконання генерується *виключення* (exception), яке надає опис помилки. Це *спеціальний клас*, який служить для передачі відомостей про помилку іншим компонентам застосування.

Щоб познайомитися з інструментами відлагодження та їх налаштуванням, напишемо просту консольну програму, яка генерує масив з десяти випадкових чисел і виводить числа у вікно консолі. У

програмі є помилка – вихід у циклі for за межі масиву. Код програми приведений в лістингу 3.1.

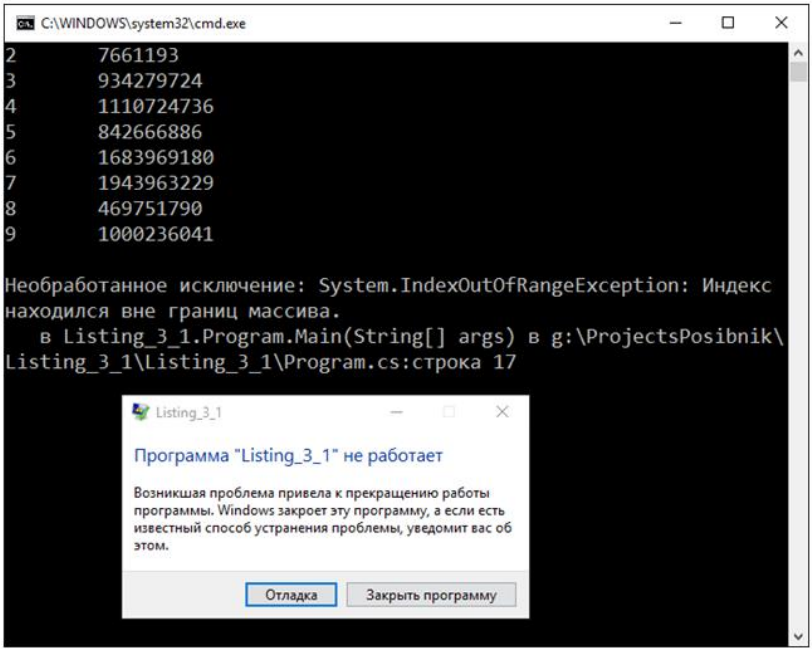
```
// Лістинг 3.1. Приклад програми з помилкою
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Listing_3_1
{
    class Program
    {
        static void Main(string[] args)
        {
                int[] numbers = new int[10];
                Random rand = new Random();
                for (int i = 0; i <= 10; i++)
                {
                        numbers[i] = rand.Next();
                        Console.WriteLine("{0}\t{1}", i, numbers[i]);
                }
                Console.Read();
        }
    }
}
```

Компіляція коду програми проходить успішно, але при виконанні застосування відлагоджувач Visual Studio реагує на помилку виходом в код і показує рядок з помилкою. На **рис. 3.3** показаний вигляд редактора Visual Studio при *генерації виключення* під час виконання програми у режимі **Отладка | Начать отладку** (клавіша **F5**).

При появі помилки в редакторі коду виділяється рядок, на якому було згенеровано помилку. Вікно у правій частині зображення – це помічник **Exception Assistant**. Він надає деталі щодо виключення і пропонує поради з пошуку помилки та усунення проблеми. З цього вікна можна отримати доступ до пошуку в

інтерактивній системі допомоги додаткової інформації щодо цього виключення.



У нижній частині головного вікна середовища розробки знаходиться ще декілька додаткових корисних вікон (**рис. 3.3**). Ліворуч розташоване вікно **Locals (Локальные)**. Це вікно відображує значення локальних змінних в коді програми у момент генерації виключення.

У правій нижній частині головного вікна середовища розробки знаходиться вікно стека викликів **Call Stack (Стек вызовов)**. Це вікно показує послідовність, в якій різні компоненти застосування були викликані. Вікно **Call Stack** можна використовувати для переходу до ділянок коду, які відображуються в стеку.

Неактивна закладка поряд з вікном **Call Stack** дає вам доступ до вікна інтерпретацій **Immediate Window (Окно интерпретации)**, яке дозволяє вводити команди коду та отримувати результати в редакторі. Далі ми детальніше розглянемо ці вікна і роботу з ними.

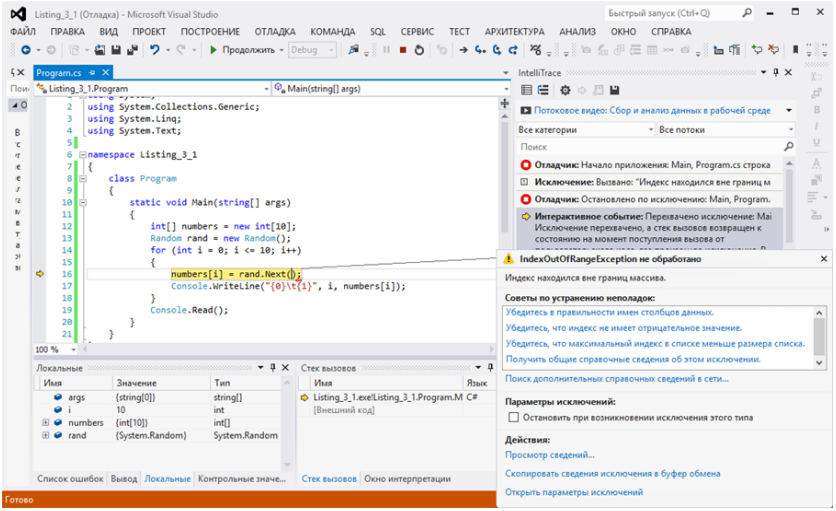


Рис. 3.3. Генерація виключення при виконанні програми

3.2. Відлагоджувач Visual Studio

Відлагоджувач, вбудований у Visual Studio, є одним з найбільших і найскладніших інструментів інтегрованого середовища розробки. Враховуючи велику функціональність, ми не зможемо описати всі можливі сценарії, які можуть зустрітися при розробці застосувань. Проте ми сподіваємося описати у цьому розділі найчастіше використовувані функціональні можливості.

Меню **Debug (Отладка)** і відповідна панель інструментів надають доступ до *запуску сеансів налаштувань, покрокового проходження коду, управління точками переривання*, а також і до багатьох функціональних можливостей налаштування у Visual Studio.

У неактивному режимі меню **Debug (Отладка)** надає можливості для запуску сеансу налаштування, прикріплення до виконуваного процесу, і для доступу до деяких вікон налаштувань.

При запуску застосування у режимі налаштування в меню **Debug** додається декілька додаткових опцій. Ці опції включають функції переміщення по коду, перезапуск сеансу і доступ до додаткових вікон налаштування.

За допомогою панелі інструментів **Debug** можна управляти сеансом налаштування. Наприклад, можна почати або продовжити сеанс налаштування, зупинити сеанс, який виконується, виконати покроковий прохід коду і т. д.

Можна управляти багатьма опціями налаштування Visual Studio в діалоговому вікні **Options (Debug | Options – Отладка | Параметры и настройки)**. Вузол **Debugging (Отладка)** в дереві опцій надає доступ до цих перемикачів налагоджувальних опцій. На **рис. 3.4** показані загальні опції налаштування в діалозі **Options**. Це вікно також доступне в головному меню в розділі **Tools (Сервис)**.

Вузол **Debugging** дає можливість включати і виключати багато опцій налаштування, наприклад, *встановлення фільтрів, точок переривання, виведення вікна попереджувальних повідомлень* та ін.

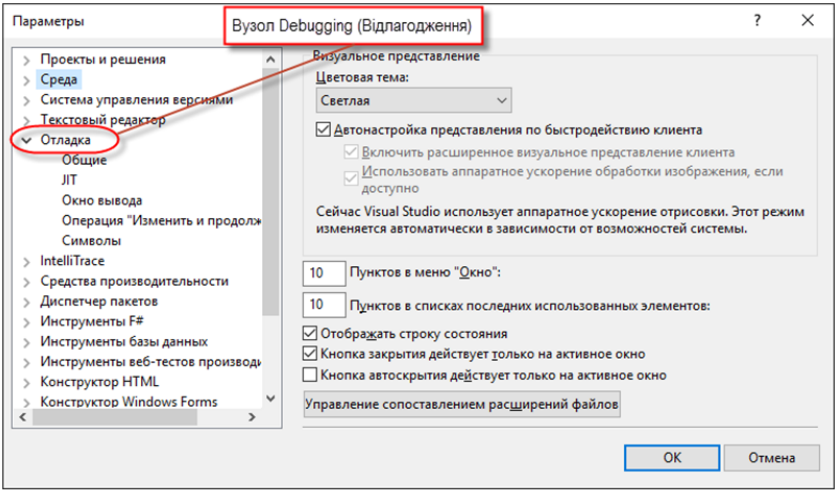


Рис. 3.4. Діалогове вікно **Options**

3.2.1. Точки переривання

Можна позначати певні рядки коду або задавати умови, при виконанні яких відлагоджувач неодмінно зупинить виконання застосування. Такі процедури називаються *встановленням точок переривання* (breakpoints), вони дозволяють зупиняти виконання

програми в заданому місці або за певних обставин. Точки переривання задаються декількома способами:

- **за функцією** – зупиняють виконання після досягнення певного рядка функції;
- **за адресою у файлі** – зупиняють виконання застосування після досягнення певного місця у файлі початкового тексту;
- **за адресою пам'яті** – зупиняють застосування при зверненні до визначеної адреси пам'яті.

Visual Studio має для точок переривання декілька позначок. *Зовнішній вигляд значків відображує різні типи точок переривання.* Наприклад, *суцільний кружок* – це звичайна точка переривання, а *порожній кружок* є деактивованою точкою переривання.

3.2.1.1. Налаштування точки переривання

Точки переривання *за функцією* застосовуються найчастіше, їх встановлюють одним з трьох способів:

1. *Клацнувши сіру область в лівій частині вікна коду навпроти потрібного рядка* – у результаті точка переривання встановлюється в цьому рядку.
2. *Клацнувши правою кнопкою потрібний рядок і вибравши* з контекстного меню команду **Insert Breakpoint** (*Вставити точку переривання*).
3. *Вибравши команду* **New Breakpoint** (*Нова точка переривання*) з меню **Debug** або з контекстного меню редактора коду і встановивши відповідні параметри точки переривання у вікні **New Breakpoint**.

Найчастіше використовуваний спосіб налаштування точки переривання: спочатку потрібно знайти рядок коду, на якому ви хочете зупинити відлагоджувач, потім ви клацаєте по цьому рядку в *полі індикаторів редактора коду*. При цьому в полі індикаторів з'являється *червоний кружок*, і *рядок коду виділяється червоним кольором* (**рис. 3.5**). Клацання мишею на кружку видаляє точку переривання.

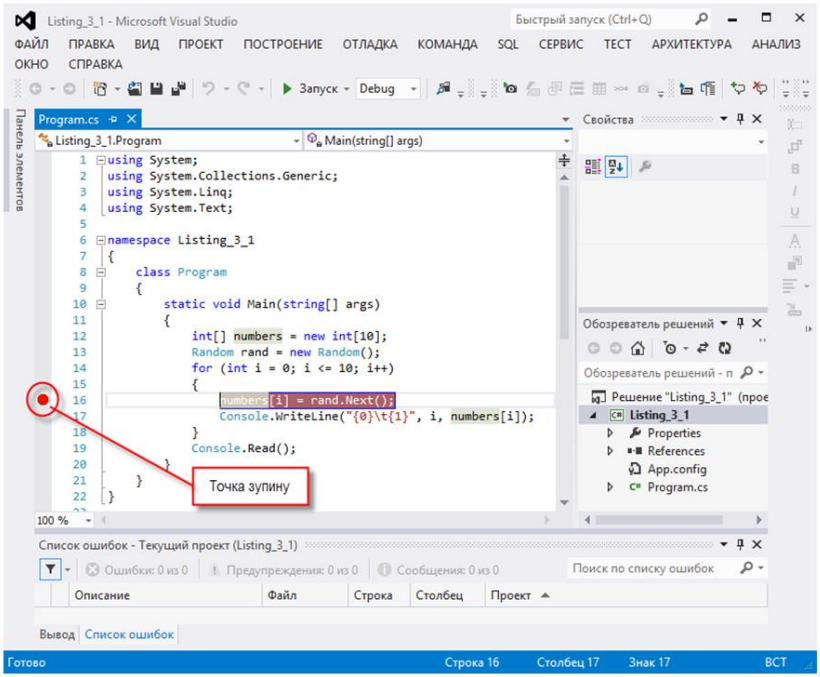


Рис. 3.5. Налаштування точки переривання

3.2.1.2 Вікно Breakpoints

Для централізованого управління всіма точками переривання застосовується вікно **Breakpoints (Точки останова)**. Його можна викликати через меню **Debug | Windows | Breakpoints**. Зовнішній вигляд вікна показаний на **рис. 3.6**.

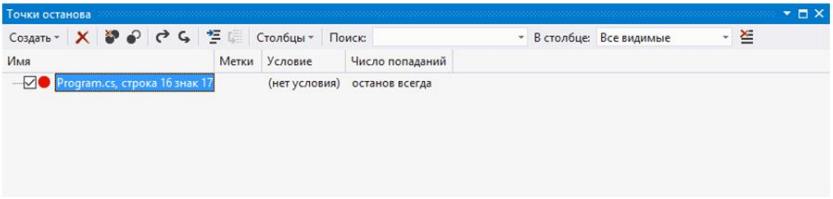


Рис. 3.6. Вікно **Breakpoints**

У вікні **Breakpoints** відображуються всі точки переривання, встановлені у проекті із зазначенням їх розміщення та умов, визначених для кожної точки переривання. У меню **Columns (Столбцы)** відображаються додаткові відомості про точки переривання (див. **рис. 3.7**).

У цьому вікні також можна деактивувати точку переривання, знявши відповідний прапорець. Крім того, кнопка у верхньому лівому кутку цього вікна дозволяє створити нові і видалити існуючі точки переривання, а також очистити список або відключити всі точки переривання.

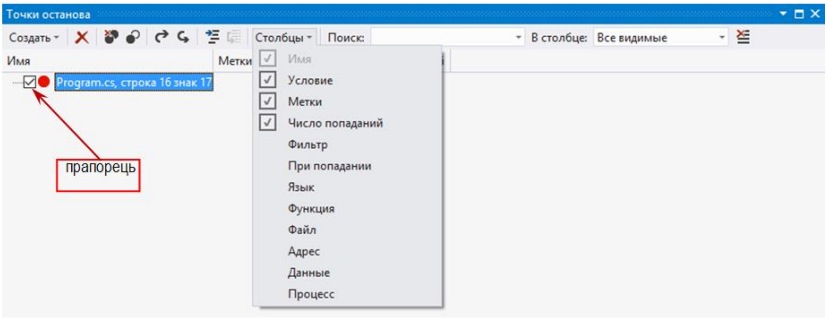


Рис. 3.7. Додаткові відомості про точки переривання у меню **Columns (Столбцы)**

Непотрібну точку переривання можна видалити або відключити, у разі її потреби у майбутньому. Як вже говорилося, для управління точками переривання призначене вікно **Breakpoints (Точки останова)**. Можна видалити або перевести в неактивний стан точку переривання в редакторі коду.

Для *видалення* точки переривання клацніть її правою кнопкою миші. Для *тимчасового відключення* точки переривання клацніть її правою кнопкою миші в редакторі коду і у контекстному меню виберіть **Disable Breakpoint (Выключить точку останова)**. Таким чином, вона буде переведена в неактивний стан.

Вікно **Breakpoints** також дає вам доступ до кожної окремої точки переривання. Воно служить стартовою точкою для налаштування багатьох опцій, пов’язаних з точками переривання. Наприклад, ви можете деактивувати окрему точку переривання,

знявши галочку біля точки переривання в списку точок переривання. Крім того, ви можете налаштувати багато властивостей та умов, пов'язаних з точкою переривання. На **рис. 3.8** показані точка відслідковування і контекстне меню, пов'язане з точкою переривання.

Зверніть увагу, що з цього контекстного меню ви можете видалити точку переривання, а також перейти до відповідного їй початкового коду. Проте більш важливе те, що тут є доступ до *налаштування умов* і *фільтрів*, пов'язаних з цією точкою переривання. Далі ми опишемо використання усіх цих опцій.

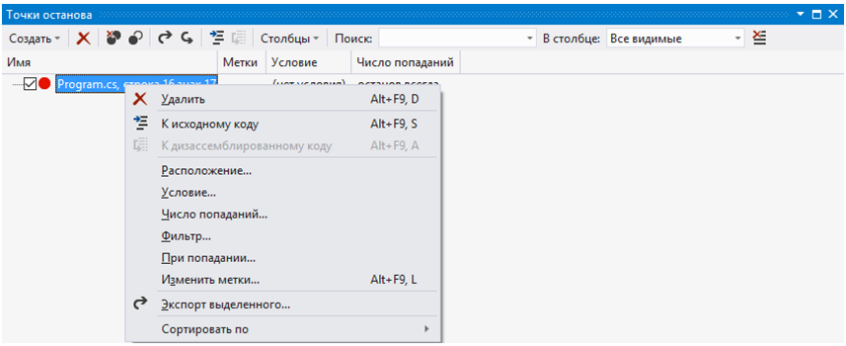


Рис. 3.8. Управління окремою точкою переривання

3.2.1.3. Налаштування точки переривання функції

Точка переривання функції – це точка, яка налаштовується в діалоговому вікні **New Breakpoint**. Вона називається *точкою переривання функції* тому, що зазвичай встановлюється на початок функції (хоча це і не обов'язково). У діалоговому вікні **Debug | New Breakpoint (Отладка | Создать точку останова | Прервать в функции…)** (див. **рис. 3.9 і 3.10**) ви можете вручну налаштувати *функцію*, на якій хочете перервати виконання, *рядок коду у функції* або навіть *символ у рядку*.

Якщо при виклику цього діалогового вікна курсор знаходиться на функції або на виклику функції, то ім'я функції буде автоматично внесене до діалогового вікна. Можна також ввести назву функції в цьому вікні.

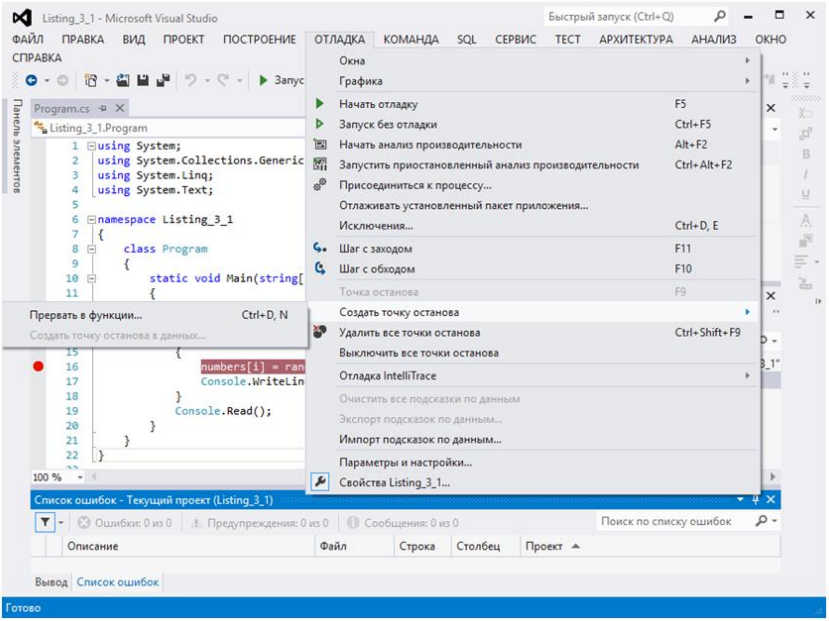


Рис. 3.9. Відкриття діалогового вікна **New Breakpoint (Отладка | Создать точку останова | Прервать в функции…)**

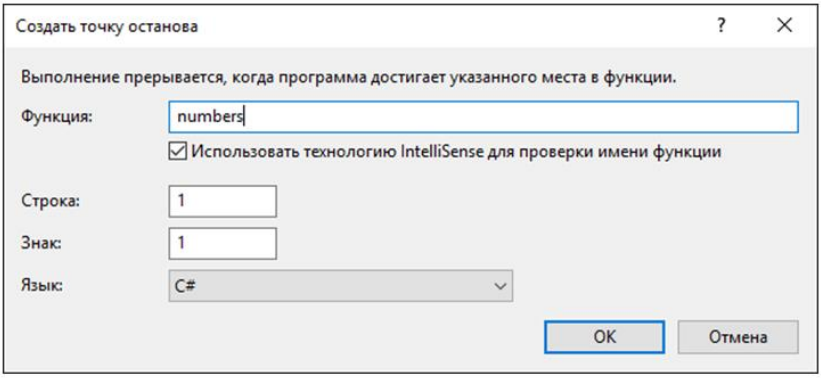


Рис. 3.10. Вікно **New Breakpoint**

3.2.1.4. Переривання на основі умов

Часто одного встановлення простої точки переривання недостатньо (або вона неефективна). Наприклад, якщо ви шукаєте виконання у вашому коді певної умови (яка, можливо, викликає винятковий стан), то вам краще здійснити переривання згідно цієї умови.

При встановленні нової точки переривання через вікно **New Breakpoint** треба задати умови, які визначають, чи буде зупинене виконання застосування після досягнення цієї точки переривання (див. **рис. 3.11**).

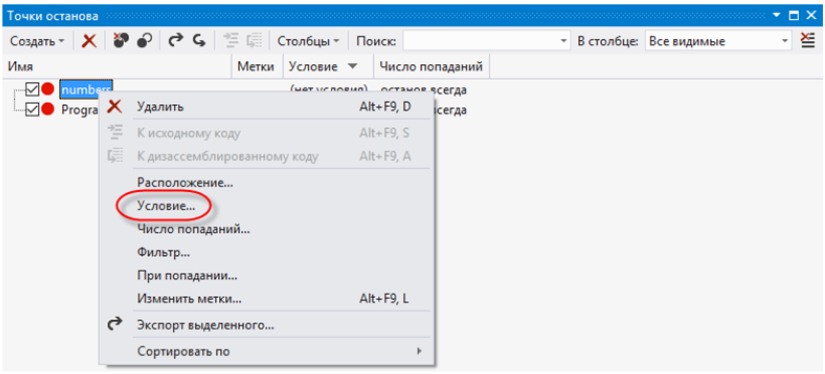


Рис. 3.11. Вікно **New Breakpoint** з контекстним меню точки переривання для вибору опції **Condition…(Условие…)**

Якщо клацнути команду **Condition…(Условие…)**, відкриється вікно **Breakpoint Condition (Условие для точки останова)**, де пропонується ввести деякий вираз (див. **рис. 3.12**). У результаті точка переривання активізується, якщо при обчисленні виразу отримаємо **true**.

Це заощадить час на постійні входження у функцію, при яких ви тільки вивчаєте декілька елементів даних і бачите, що ваша умова не виконана. Є декілька типів умов, які ви можете додати до точки переривання.

Всі типи умов для точок переривання встановлюються за допомогою спеціалізованих вікон, які ми детально розглянемо далі.

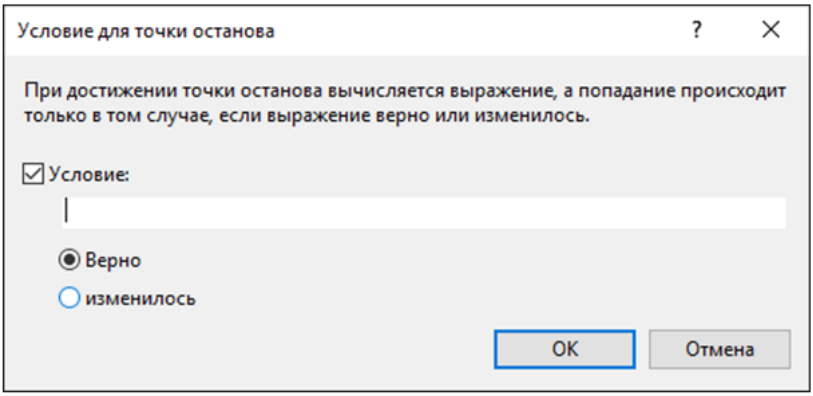


Рис. 3.12. Вікно **Breakpoint Condition (Условие для точки останова)**, де пропонується ввести деякий вираз

3.2.1.5. Вікно File Breakpoint

Можна відредагувати точку переривання так, щоб переривання відбувалося у визначеному місці файла. Більшість точок переривання працюють саме так. Тобто вони знають файл, номер рядка і символ, на якому повинне статися переривання виконання коду програми.

На **рис. 3.13** показаний приклад вікна **File Breakpoint (Точка останова в файле)**, відкритого через опцію **Location… (Расположение…)** *контекстного меню точки переривання*. *Можна також використовувати* цю функціональну можливість для швидкого встановлення точки переривання у конкретному рядку без потреби пошуку в коді програми.

3.2.1.6. Вікно Breakpoint Condition

Умова точки переривання дозволяє при виконанні програми вийти у відлагоджувач тоді, коли повинна виконатися деяка умова. Наприклад, якщо відомо, що помилка в програмі з'являється тільки при певних значеннях змінної. Налаштування умови точок переривання – це *зручний засіб для пошуку* таких помилок.

Для налаштування умови виділіть точку переривання, для якої ви хочете додати умову. Потім виберіть опцію **Condition** з контекстного меню (клацанням правою кнопкою миші). Це активує діалогове вікно **Breakpoint Condition** (**рис. 3.12**). При налаштуванні

умови у вас є два варіанти: **Is true (Верно)** і **Has changed (Изменилось)**. Варіант **Is true** дозволяє вам налаштувати логічну умову, при виконанні якої станеться вихід з відлагоджувача на відповідний рядок коду.

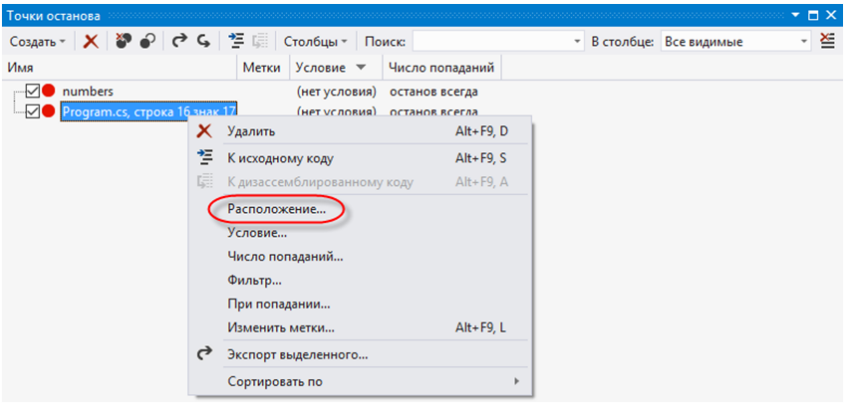


Рис. 3.13. Вікно **New Breakpoint** з контекстним меню точки переривання для вибору опції **Location… (Расположение…)**

Наприклад, для застосування з **лістингу 3.1** можна додати **i=5** в умову **Is true** для точки переривання (де **i** – змінна циклу **for**). Тоді відлагоджувачу буде дано вказівку зупинятися на цьому рядку коду тільки тоді, коли буде виконано умову **i=5**. На **рис. 3.15** показано цю умову в діалоговому вікні. Там же є і дві опції для умов.

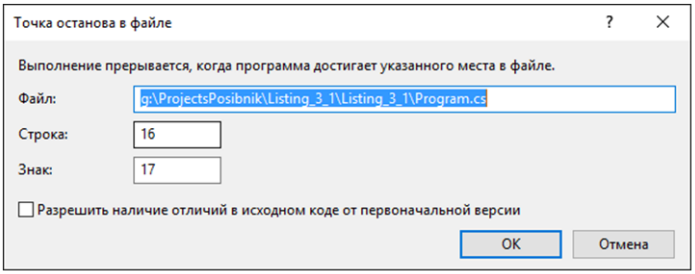
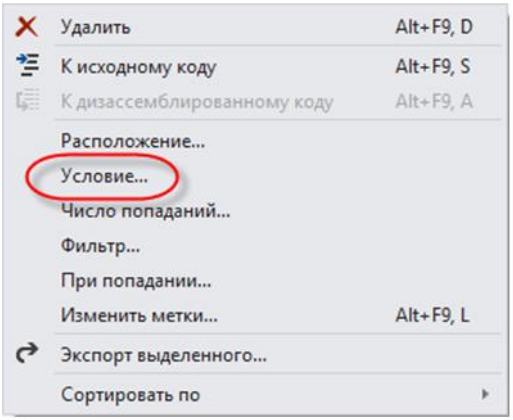


Рис. 3.14. Діалогове вікно **File Breakpoint (Точка останова в файле)**



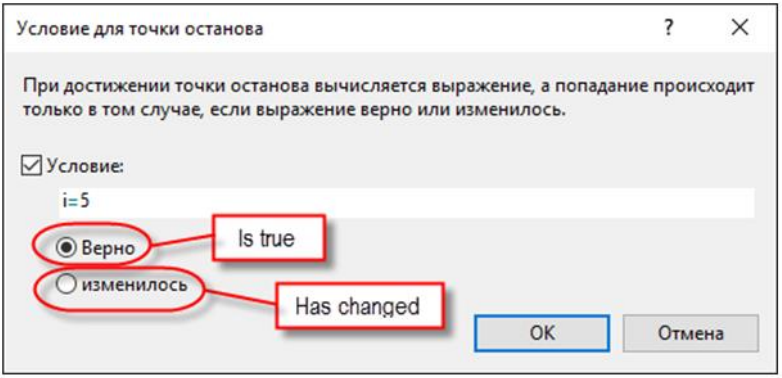


Рис. 3.15. Діалогове вікно **Breakpoint Condition**

Опція **Has changed** означає, що *виходити з коду треба тільки при зміні значення виразу*. Перший прохід по коду встановлює перше значення виразу. Якщо після цього це значення змінюється, то відлагоджувач робить переривання в цьому рядку. Ця можливість *може бути корисною* тоді, коли у вас є поля або властивості з початковими значеннями і ви хочете відстежити зміни цих значень. Крім того, опція **Has changed** *може бути корисна* в циклах та операторах **if..then**, коли вас цікавить тільки одне – чи змінив ваш код це конкретне значення.

Інформація про точки переривання між сеансами відлагодження зберігається. Тобто коли у кінці робочого дня ви закриваєте Visual Studio, то повернувшись, всі точки переривання виявляться на своїх місцях. Це окупає час, витрачений на налаштування складних режимів відлагодження. Вони можуть залишатися у вашому застосуванні і включатися (вимикатися) в міру потреби.

3.2.1.7. Вікно Breakpoint Hit Count

Використовуючи команду **Hit Count (Число попаданий)**, ви повідомляєте відлагоджувач, що хочете перервати виконання тоді, коли цей рядок коду *виконається певне число разів*. Зазвичай можна знайти зручнішу умову переривання, чим **Hit Count**. Однак ця *функція корисна* в тих випадках, коли не можна вказати реальну умову, але відомо, що при проходженні через функцію певне число разів, починаються проблеми. Крім того, опція **Hit Count** може бути кориснішою в сценаріях з точками відслідковування, коли ви виводите дані про те, що відбувається у вашому коді. Можливо, вам буде зручно виводити ці дані тільки час від часу.

На **рис. 3.16** показано діалогове вікно **Breakpoint Hit Count**. Зверніть увагу, що цей моментальний знімок екрану був зроблений під час активного сеансу налаштування. Ви можете додати будь-які з цих умов до точок переривання в час активного сеансу налаштування.

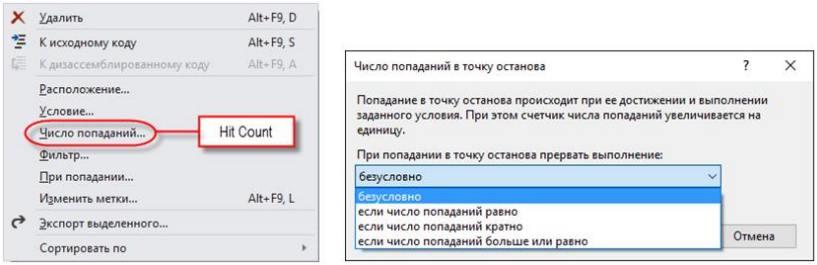


Рис. 3.16. Налаштування числа проходжень точки переривання

Це діалогове вікно надає вам декілька опцій для налаштування числа попадань. У розкривному списку **Whenthe breakpoint is hit** є наступні варіанти:

- **break always** – переривати завжди, не використовує лічильник числа попадань (за замовчуванням);
- **break when the hit count is equal to** – переривати тоді, коли значення лічильника дорівнює вказаному числу;
- **break when the hit count is a multiple of** – переривати тоді, коли значення лічильника кратне вказаному числу;
- **break when the hit count is greater than or equal to** – переривати тоді, коли значення лічильника більше або дорівнює вказаному числу.

При потребі можна спільно використовувати всі умови для точок переривання в одній точці переривання. Наприклад, додати до цієї точки переривання умову і фільтр, що дає можливість *створювати за допомогою точок переривання спеціальні сценарії* для налаштування застосування.

3.2.1.8. Вікно Breakpoint Filter

Середовище розробки надає діалогове вікно **Breakpoint Filter** (**Фильтр…**). *Фільтри точок переривання* дозволяють вам вказати конкретний комп'ютер, процес або потік, в якому повинна статися зупинка виконання програми. Наприклад, якщо помилка з'являється тільки на визначеному комп'ютері, в певному процесі або потоці, то є можливість налаштувати таке переривання за допомогою фільтра.

Для налаштування фільтрів точок переривання можна задати ім'я цільового комп'ютера або для процесу задати його ім'я, або ідентифікатор. Є можливість створювати комбінації умов за допомогою операцій **&** (*і*), **||** (*або*) та **!** (*ні*). Тонке налаштування фільтрів точок переривання дозволяє встановити переривання у визначеному потоці конкретного процесу на конкретному комп'ютері. На **рис. 3.17** показано діалогове вікно **Breakpoint Filter**, в якому можна налаштовувати потрібні фільтри точок переривання.

3.2.1.9. Вікно When Breakpoint Is Hit

Точки відслідковування дозволяють вивести дані у вікно **Output (Вывод)** (це вікно буде описано далі), коли зустрілася певна точка переривання. Після цього ви можете вийти у відлагоджувач, як і у випадку із звичайною точкою переривання, або продовжити виконання застосування. Ця можливість може бути дуже корисною тоді, коли ви хочете підтримувати реєстрацію подій, які відбуваються в застосуванні при налаштуванні. Потім ви можете переглянути журнал подій, щоб отримати інформацію про конкретні умови і порядок виконання (коли відбувається виключення).

Ви можете налаштувати точки відслідковування безпосередньо – клацнувши правою кнопкою миші по рядку коду і наступного вибору пункту **Insert Tracepoint** в меню **Breakpoint** (див. **рис. 3.18**).

Крім того, вибір команди **When Hit** з контекстного меню точки переривання у вікні **Breakpoints** активує діалогове вікно точки відслідковування **When Breakpoint Is Hit** (рис. **3.19**).

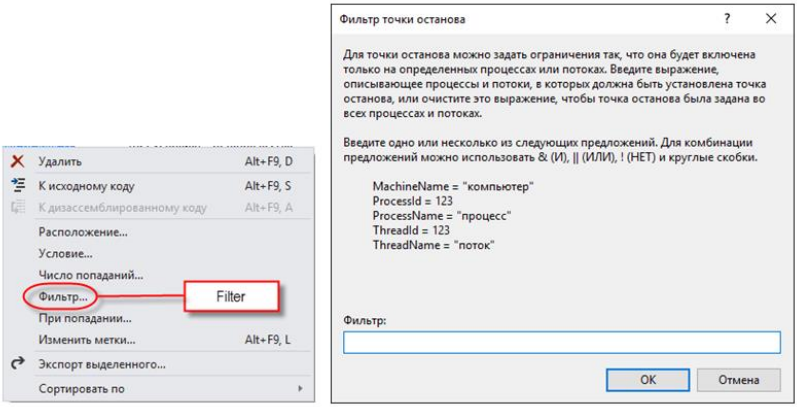


Рис. 3.17. Вікно **Breakpoint Filter**

Наявні в діалоговому вікні **When Breakpoint Is Hit** опції включають виведення повідомлення у вікно виводу, виконання макросу і продовження виконання. Можна вибрати будь-яку комбінацію цих опцій. Перша (вивід повідомлення) дозволяє вивести

дані про функцію. Є декілька ключових слів, які допускається використовувати для виведення даних, наприклад:

- $function – для імені функції;
- $caller – для імені викликаючої функції.

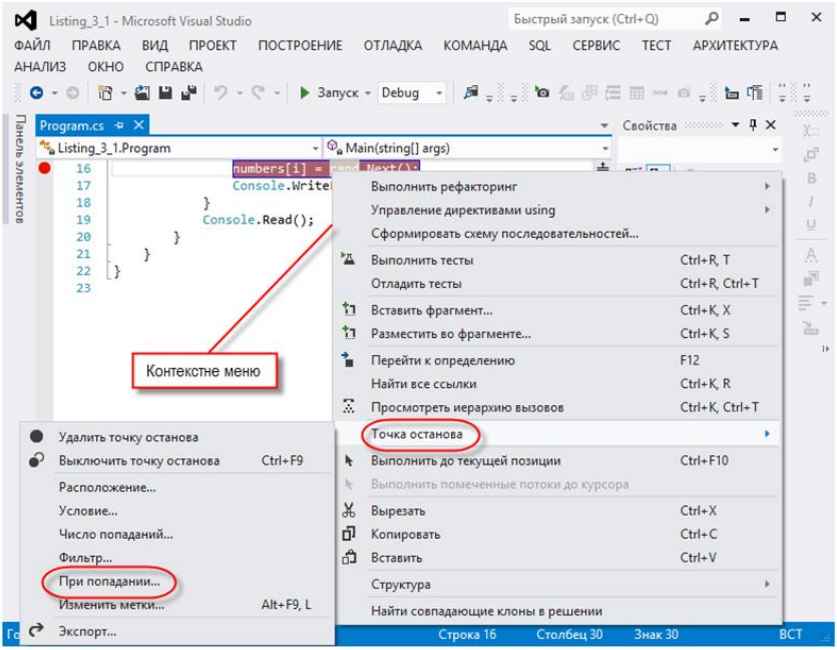


Рис. 3.18. Відкриття вікна **Is Hit… (При попадании…)**, використовуючи контекстне меню рядка коду

Повний список ключових слів відображується у пояснюючому тексті вікна **When Breakpoint Is Hit**. Можна також вивести значення змінних, взявши імена змінних у фігурні дужки.

Прапорець **Continue execution (Продолжить выполнение)** дозволяє вказати, чи дійсно це справжня точка відслідковування чи це точка переривання, яка містить дію з відслідковування. Якщо ви вибираєте *продовження виконання*, то отримуєте тільки відслідковуючу дію (повідомлення і/або макрос). Якщо ж ви даєте вказівку, що *продовжувати виконання програми не треба*, то

отримуєте відслідковуючу дію плюс зупинку відлагоджувача на цьому рядку коду так само, як і на звичайній точці зупину. При цьому, по суті, застосовується дія **When Hit** до звичайної точки переривання.

Також є можливість об'єднувати дії відслідковування з умовами. При цьому дія відбувається тільки у тому випадку, коли виконується умова точки переривання.

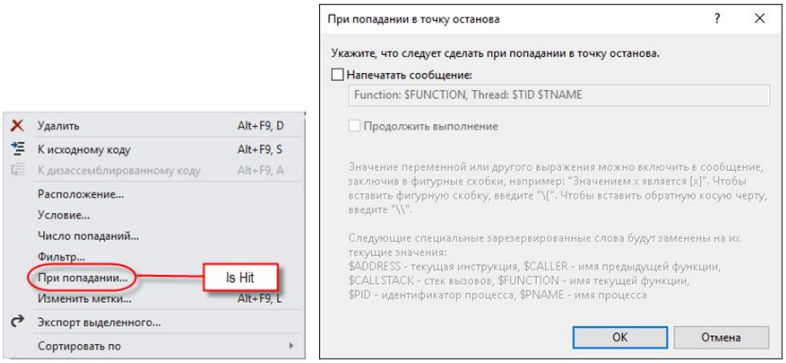


Рис. 3.19. Вікно **When Breakpoint Is Hit**

3.3. Покрокове проходження коду

При налаштуванні програми найчастіше використовується покроковий прохід рядками коду і вивчення даних, які виводяться застосуванням і відлагоджувачем. Покрокове проходження полягає у *вивченні рядка коду, виконанні цього рядка* та *аналізі результатів виконання*. Далі цей процес повторюється для наступних рядків програмного коду.

3.3.1. Початок налаштування застосування

Команда входу в покрокове проходження коду **Step Into (Шаг с заходом)** доступна в меню **Debug (рис. 3.20)** і на панелі інструментів (ця команда також виконується при натисненні клавіші **<F11>**).

Якщо виконати команду **Step Into** для застосування, *яке в цей момент не виконується в режимі налаштування*, застосування буде

відкомпільоване і запущене, а у вікні відлагоджувач ви отримаєте перший рядок для покрокового проходження коду. Це і є вхід в код вашого застосування. На **рис. 3.21** показане застосування в режимі налаштування при виклику команди **Step Into**.

Виклик команди **Step Over (Шаг с обходом)** з меню **Debug** у той момент, коли застосування знаходиться в стані спокою, приведе до того ж самого, що і виклик **Step Into**. Тобто застосування відкомпілюється і запуститься в сеансі налаштування на перший рядок коду.

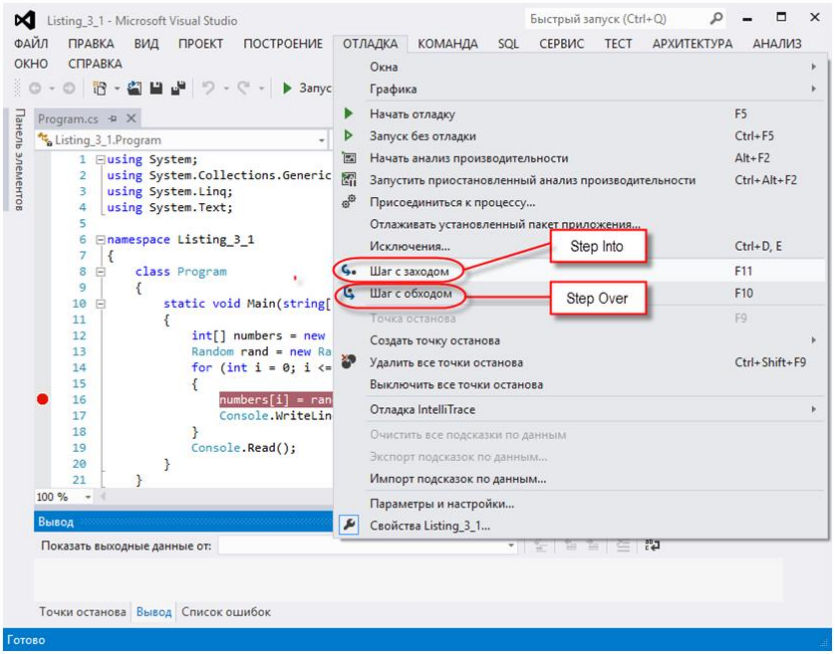


Рис. 3.20. Команда **Step Into** в меню **Debug**

Однією із зручних функціональних можливостей набору інструментів налаштування є функція **Run to current position (Выполнить до текущей позиции)**. Якщо встановити курсор на конкретний рядок в коді програми, викликати контекстне меню і виконати команду **Run to current position**, застосування відкомпілюється і виконуватиметься доти, поки не дійде до того

рядка коду, де був встановлений курсор (див. **рис. 3.22**). У цій точці відлагоджувач перериває застосування і виділяє рядок коду для виконання покрокового проходження згідно коду програми (див. **рис. 3.23**). Тому функція **Run to current position** є ефективним засобом привести відлагоджувач на вказану програмістом область коду, яка вимагає налаштування. Фактично це невидима тимчасова точка переривання.

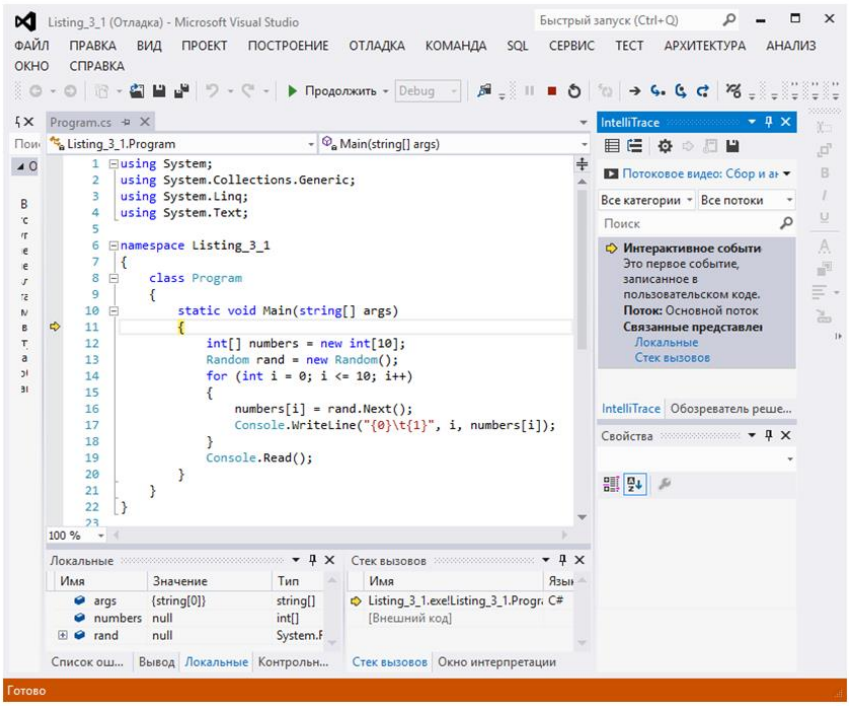


Рис. 3.21. Використання **Step Into** для запуску застосування

Сеанс налаштування застосування можна також запустити командою **Start Debugging** (**Запуск**). Це *зелена стрілка* в меню **Debug** або на панелі інструментів. Налаштування застосування також можна запустити натисненням клавішу **<F5>**. При цьому почнеться сеанс налаштування, але виходу в код не станеться, якщо тільки при виконанні програми не буде згенероване виключення або не

трапиться точка переривання. Ця операція використовується, якщо в програмному коді застосовується велике число точок переривання.

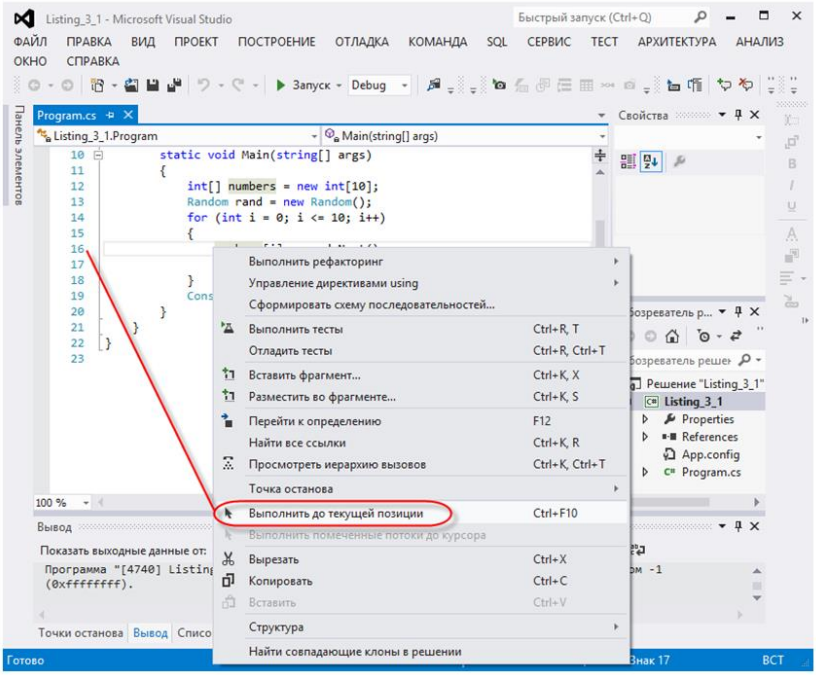


Рис. 3.22. Контекстне меню з командою **Run to current position (Выполнить до текущей позиции)**, з курсором встановленим на **16**-й рядок

Якщо ваше застосування виконується, і ви хочете увійти до режиму переривання, то ви можете зробити це в будь-який час або за допомогою команди **Break All** (**Прервать все**) з меню **Debug,** або панелі інструментів, або за допомогою комбінації клавіш **<Ctrl>+<Alt>+<Break>**. Команда **Break All** представлена на панелі інструментів значком з *символом паузи* – ⏸ . Натиснення цієї кнопки зупиняє ваше застосування на наступному виконуваному рядку, і дозволяє вам отримати інформацію з відлагоджувача. Команда **Break All** *особливо корисна* у тому випадку, коли вимагається перервати тривалий процес або цикл в коді програми.

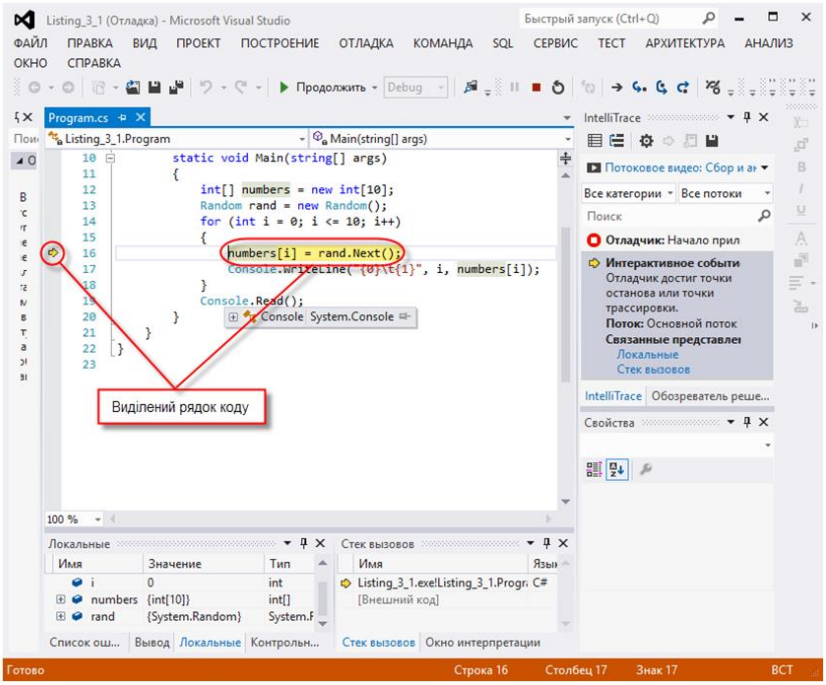


Рис. 3.23. Переривання відлагоджувачем виконання застосування і виділення рядка коду для виконання покрокового проходження згідно коду програми

3.3.2. Проходження по коду

Під час сеансу налаштування є три основні варіанти навігації по коду. Можна увійти до рядка або функції, пропустити цю функцію, або вийти з функції. Розглянемо кожний із них:

- Команда **Step Into** – **<F11>** – дозволяє рухатися по коду рядок за рядком. Виклик цієї команди виконає поточний рядок коду і помістить курсор на наступний виконуваний рядок. Важлива відмінність між **Step Into** та іншими схожими командами полягає в тому, як **Step Into** *обробляє рядки коду, в яких знаходяться виклики методів*. Якщо ви знаходитеся на рядку коду, який викликає інший метод

рішення, то виклик **Step Into** перенесе вас на перший рядок цього методу (за умови, що у вас завантажені відповідні налагоджувальні символи).

- Команда **Step Over** – **<F10>** – дозволяє *зберігати фокус в поточній процедурі* (не заходячи в методи, які викликаються нею). Тобто виклик **Step Over** приведе до виконання рядка за рядком, але *не заведе вас у виклики функцій, конструктори* або *виклики властивостей*. Звичайно, будь-яке виключення в пропущеній функції приведе до переривання роботи відлагоджувача і виходу в код (як завжди).
- Команда виходу з покрокового режиму **Step Out**, яка викликається натисканням комбінації клавіш **<Shift>+<F11>** – це ще один корисний інструмент. Він дозволяє вам дати вказівку відлагоджувачу закінчити виконання поточного методу (який ви налаштовуєте) і повернутися в режим переривання виконання програми відразу після його завершення. Це дуже зручно тоді, *коли треба пропустити великий фрагмент коду цього методу, який написаний без помилок* (або всі помилки усунені) і не вимагає покрокового виконання. Крім того, у разі потреби можна увійти до функції для налаштування тільки її частини, а потім вийти з неї. Зауважимо, що ця команда з'являється в меню **Debug**, *коли застосування знаходиться в режимі налаштування* (див. **рис. 3.24**).

3.3.3. Продовження налаштування

Якщо в сеансі налаштування ви покинули виконуваний код, то знайти дорогу назад часто буває дуже важко. Рядок, який виконувався, загубився в одному з багатьох відкритих вікон з кодом. На щастя, для того, щоб повернутися назад, ви можете використати кнопку **Show Next Statement** (*значок з жовтою стрілкою* (див. **рис. 3.24**, **рядок 11**)) на панелі інструментів **Debug**. Це поверне вас на той рядок, який виконувався у момент зупинки відлагоджувача.

Коли програма знаходиться в сеансі налаштування, команда **Start Debugging (Начать отладку)** змінюється на **Continue (Продолжить)**. Команда **Continue** доступна тоді, коли ви припинили виконання на рядку коду у відлагоджувачі (див. **рис. 3.25**). Ця

команда дає можливість продовжити виконання застосування без покрокового проходження по рядках коду програми. За допомогою команди **Continue** ви даєте вказівку відлагоджувачу продовжувати виконання програми доти, поки не станеться виключення або не спрацює точка переривання.

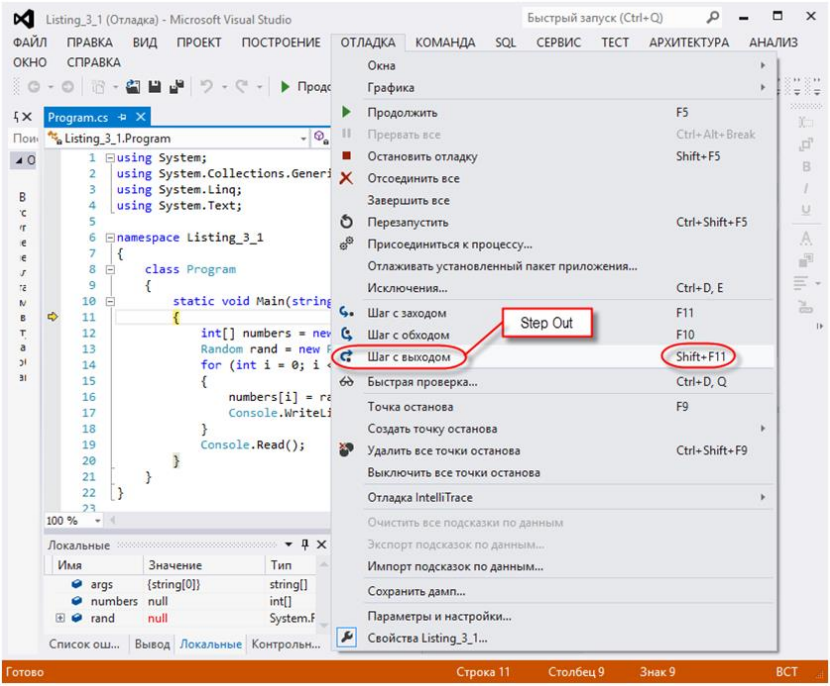


Рис. 3.24. Команда **Step Out** в меню **Debug**

3.3.4. Закінчення налаштування

Ви можете закінчити сеанс налаштування декількома способами. Один із найчастіше використовуваних методів – це припинити виконання застосування. При завершенні застосування відбудеться також і завершення сеансу налаштування.

Є також пара способів і у вікні **Debug**. Команда **Terminate All (Завершить все)** *завершує всі процеси*, до яких прикріплений відлагоджувач, і завершує сеанс налаштування. Є також опція **Detach All (Отсоединить все)**, яка *відкріпляє відлагоджувач* від усіх

виконуваних процесів, *без їх завершення*. Опція **Detach All** потрібна у разі *тимчасового прикріплення до виконуваного процесу*, який повинен залишитися працюючим, і після завершення налаштування.

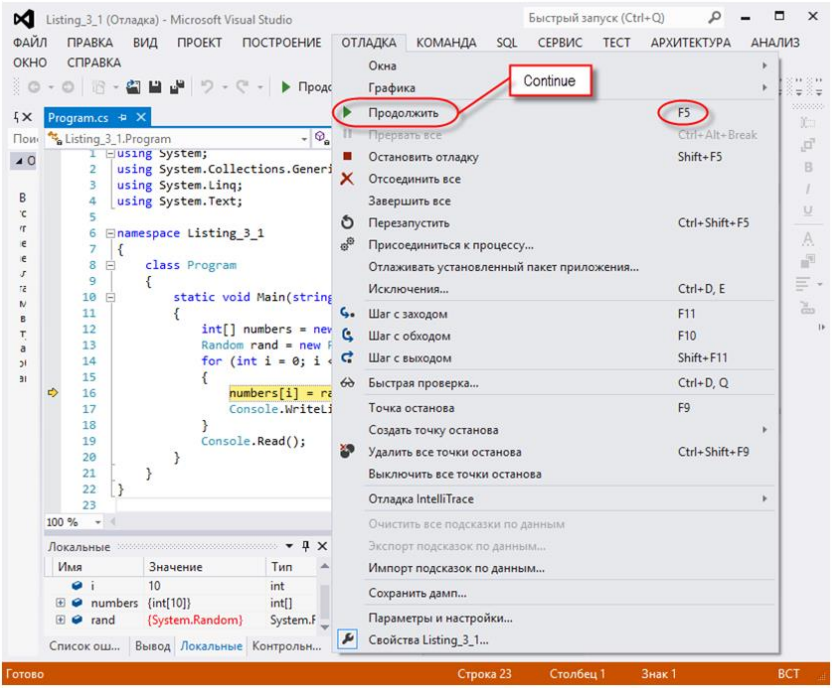


Рис. 3.25. Команда **Continue** в меню **Debug**

3.4. Налагоджувальні вікна

Середовище VS надає багато вікон, які дозволяють відслідковувати виконання програми. Знаходження потрібних даних допоможе вам швидко виявити помилки і швидко їх виправити. Visual Studio надаватиме вам дані там, де вони потрібні. Наприклад, вікно спливаючої підказки **DataTips** показує значення змінних прямо в редакторі коду. Є ще багато прикладів того, як VS показує налагоджувальні дані тоді і там, де вони вам треба. У наступних розділах ми опишемо ці випадки (і не лише їх). Одні з них, такі як вікна **Output** і **Command**, *доступні в режимі* **Design**

(Конструирование), а інші, наприклад вікно **Locals**, *видні тільки при налаштуванні*.

Найчастіше використовувані при налаштуванні вікна **Locals, Autos** і **Watch** дозволяють відстежувати і редагувати значення змінних в коді програми, що зручно при тестуванні процедур шляхом передачі їм різних аргументів.

Для демонстрації використання налагоджувальних вікон скористаємося нескладною програмкою з помилкою (**рис. 3.26**) (у **розділі 3.13** ми розглянемо складніший приклад докладніше). Відлагоджувач виявив помилку і повідомляє про це з допомогою спливаючого вікна. Проаналізуйте програму і виявіть помилку до запуску програми з метою налаштування.

Переглянемо вміст вікон налаштування.

3.4.1. Вікно Output

У вікні **Output (Вывод)** відображуються всі дані, які застосування виводить в командному рядку під час його компіляції і виконання: *оператори* **Debug** і **Trace** (про ці оператори буде розказано далі), а також *повідомлення*, якими застосування повідомляє про завантаження збірок. При налаштуванні застосування у вікні **Output**, зазвичай, переглядають повідомлення оператора **Debug**. За замовчуванням вікно **Output** видно, т. т. воно присутнє при завантаженні; інакше його можна викликати, вибравши з меню команду **View | Other Windows | Output** (**рис. 3.27**).

3.4.2. Вікно Locals

Під час *виконання програми в режимі налаштування* у вікні локальних змінних **Locals (Локальные)** можна відслідковувати значення всіх змінних поточної процедури. У ньому відображуються три стовпці: **Name** (**Имя**), **Value** (**Значение**) і **Type** (**Тип**).

Щоб відобразити вікно **Locals** під час налаштування застосування, потрібно вибрати в меню **Debug** команду **Windows | Locals**.

Вікно **Locals** показує всі змінні та їх значення для *поточної області видимості* відлагоджувача і, отже, можна отримати інформацію про те, що *відбувається у поточному методі*. Змінні у вікні **Locals** автоматично налаштовуються відлагоджувачем VS. Ці

змінні *організовані в список і відсортовані* за іменем в алфавітному порядку.

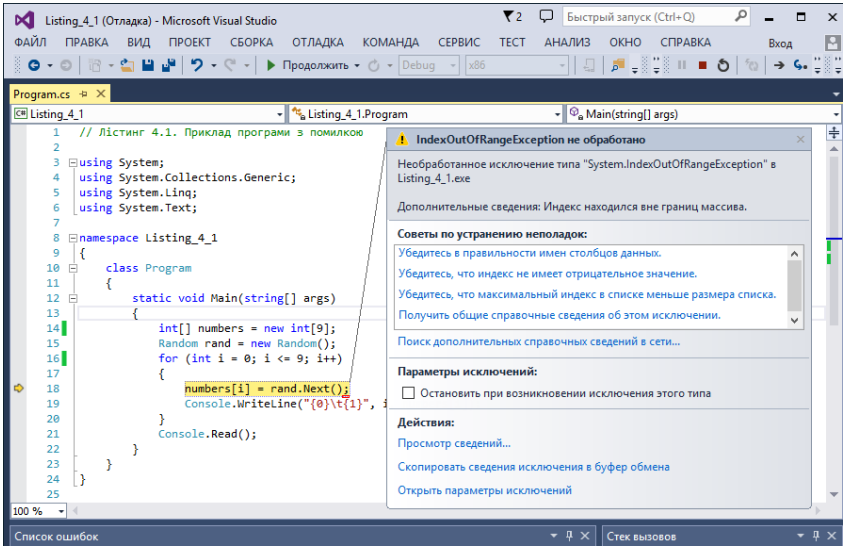


Рис. 3.26. Приклад програми з помилкою

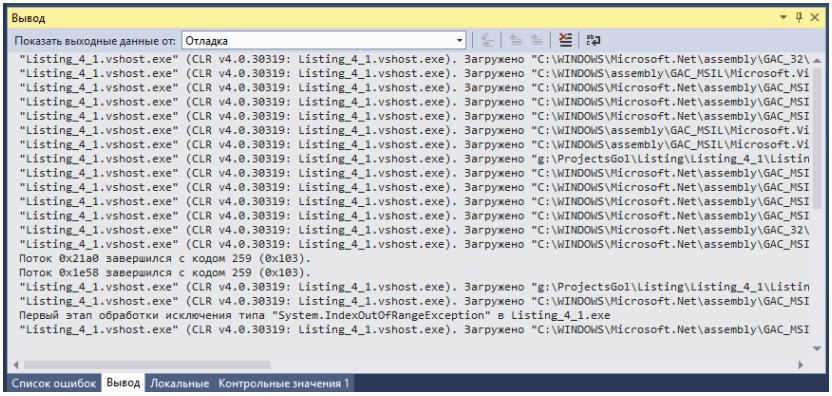


Рис. 3.27. Вікно **Output**

Складні змінні, наприклад екземпляри класів і структур, показані у вигляді дерева з вузлами, розкриваючи які можна отримати доступ до значень їх членів.

При передачі управління від методу до методу вміст вікна **Locals** міняється, *відображуючи тільки локальні змінні*.

У вікні **Locals** також можна змінювати значення змінної. Для цього треба виділити в стовпці **Value (Значение)** значення потрібної змінної і ввести нове значення. Проте у такий спосіб можливе коригування тільки значень *для простих типів*, наприклад рядкових і числових змінних. Модифікувати змінні-об'єкти, поміщаючи в них посилання на інші екземпляри класу або структури, не можна, зате можна змінювати значення їх членів. Змінені значення в стовпці **Value** підсвічують червоним кольором.

На **рис. 3.28** показаний приклад вікна **Locals**. У ньому ви можете бачити застосування нашого прикладу, яке припинене всередині циклу **for**. У міру встановлення значень результати відображуються у стовпці **Value** вікна **Locals**.

Рис. 3.28. Вікно **Locals**

3.4.3. Вікно Autos

Вікно **Autos (Видимые)** нагадує вікно **Locals**, але його можливості обмежені. У ньому відображуються ті ж стовпці, що і у вікні **Locals**, але в них *містяться тільки змінні з поточного і попереднього рядків коду*. У вікні **Autos** можна змінювати значення змінних так само, як і в **Locals**.

Щоб викликати вікно **Autos** *під час налаштування застосування*, виберіть з меню **Debug** команду **Windows | Autos**.

Використовувати вікно **Autos** зручно при налаштуванні великих програм, коли відображення всіх локальних змінних дає надто багато інформації, щоб в ній можна було розібратися. Це вікно *відображує значення всіх змінних і виразів*, наявних в *поточному*

виконуваному рядку коду, або в *попередньому* рядку коду. Це дозволяє вам дійсно зосередитися тільки на значеннях, які ви в даний момент налаштовуєте.

На **рис. 3.29** показано вікно **Autos** для рядка коду. Зверніть увагу на різницю між вікнами **Locals** та **Autos**, а також на те, що середовище Visual Studio навіть додало в список спостереження специфічні вирази, які відрізняються від коду.

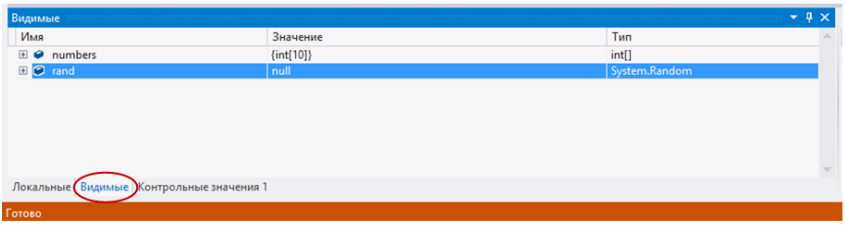


Рис. 3.29. Вікно **Autos**

3.5. Функція налаштування IntelliTrace

IntelliTrace можна використовувати для *збору відомостей* про *конкретні події* або *категорії подій*, а також про *окремі виклики функцій* на додаток до подій. Процедура приведена нижче.

Щоб записувати і відстежувати історію виконання коду або застосування в **IntelliTrace** у Visual Studio Ultimate, досить розпочати налаштування, як завжди. За замовчуванням **IntelliTrace** включається та автоматично записує конкретні події і дані, щоб ви могли бачити, що відбувається у вашому застосуванні.

3.5.1. Налаштування збору даних IntelliTrace для відлагодження у VS

Щоб контролювати обсяг інформації, який відображається під час налаштування з використанням **IntelliTrace** у VS можна змінити налаштування колекції **IntelliTrace**. Налаштування **IntelliTrace** застосовуються до усіх проектів і рішень. Вони залишаються незмінними у всіх сеансах налаштування і сеансах VS.

3.5.1.1. Налаштування колекції подій

Можна *включати* або *відключати колекцію* конкретних *подій* **IntelliTrace** за винятком подій відлагоджувача і виключень, які завжди збираються. *Події* **IntelliTrace** – це події відлагоджувача, виключення, події .NET Framework та інші системні події, які допоможуть вам у процесі налаштування. Додаткові відомості див. в розділі **Налаштування застосування шляхом запису виконання коду за допомогою IntelliTraceза адресою: https://msdn.microsoft.com/ru-ru/library/dd264915(v=vs.110).aspx.**

1. Переконайтеся, що **IntelliTrace** включений (див. **рис. 3.30**).

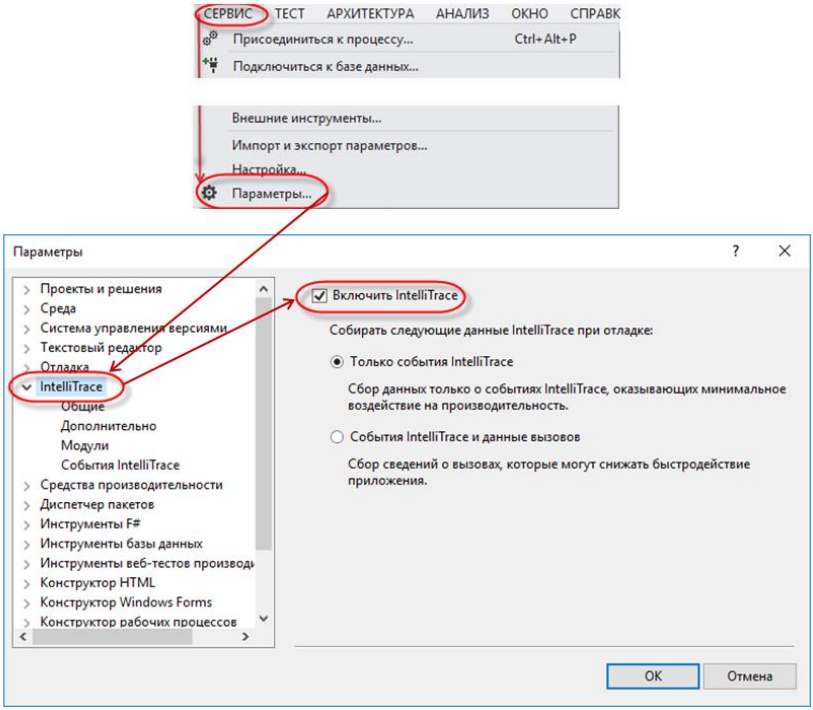


Рис. 3.30. Схема включення **IntelliTrace**

Увага: Зняття прапорця **Включить IntelliTrace** відключає всі параметри користувача.

2. Виберіть *події* і *категорії подій*, які треба записувати (див. **рис. 3.31**).

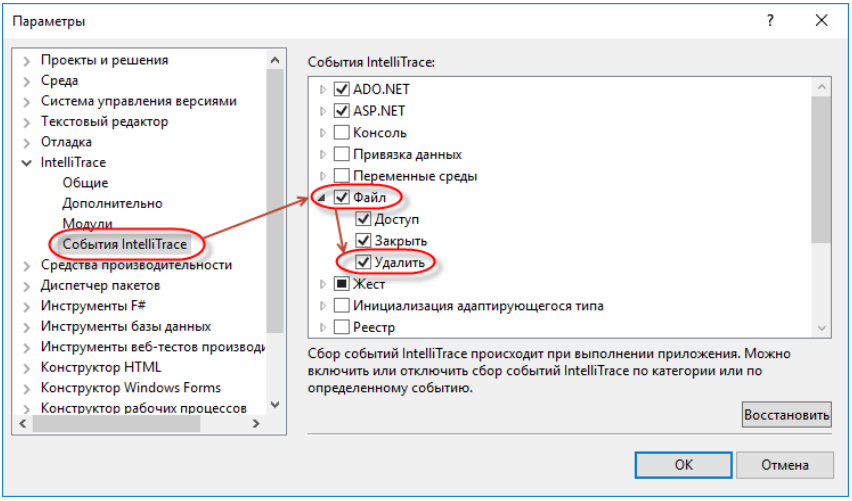


Рис. 3.31. Вибір подій і категорій подій **IntelliTrace**

3. Якщо треба, запустіть повторно сеанс налаштування. Щоб розпочати налаштування за допомогою **IntelliTrace**, **див. розділ Запис виконання коду за допомогою IntelliTrace для налаштування у Visual Studio за адресою: https://msdn.microsoft.com/ru-ru/library/dd572114(v=vs.110).aspx**.

Дані, які **IntelliTrace** збирає для подій **IntelliTrace**.

- **Події відлагоджувача**
  Щоб уникнути проблем з продуктивністю **IntelliTrace** не записує усі можливі значення події відлагоджувача. Замість цього він записує ці значення:
  - *Значення у вікні* **Локальные**. Залишіть вікно **Локальные** відкритим, щоб бачити ці значення.
  - *Значення у полі* **Видимые**, тільки якщо відкрите вікно **Видимые.**

- *Значення підказок даних*, які відображаються при наведенні покажчика миші, на змінну у вікні джерела для перегляду його значення. **IntelliTrace** не збирає значення в закріплених підказках даних.

- **Виключення**
  IntelliTrace записує тип виключення і повідомлення для відповідних типів виключень.
  - Оброблені виключення, якщо виключення створене і перехоплене.
  - Необроблені виключення.
- **Події .NET Framework**
  За замовчуванням **IntelliTrace** записує найбільш розповсюджені події платформи .NET Framework. Наприклад:
  - При події доступу до файла **IntelliTrace** записує ім'я файла.
  - Для події встановлення прапорця **IntelliTrace** записує стан і текст прапорця.

3.5.1.2. Налаштування колекції викликів функцій

Якщо колекція викликів включена, можна перемкнутися з традиційного налаштування на налаштування за допомогою **IntelliTrace**, щоб покроково перевірити код і переглянути історію стека викликів. Для включення цієї можливості включіть колекцію викликів перед початком сеансу налаштування. Додаткові відомості **див. в розділі Налаштування застосування шляхом запису виконання коду за допомогою IntelliTrace за адресою: https://msdn.microsoft.com/ru-ru/library/dd264915(v=vs.110).aspx**.

1. Включіть або вимкніть **События IntelliTrace** і відомості про виклик (див. **рис. 3.32**).
2. При потребі повторно запустіть сеанс налаштування. Щоб розпочати налаштування за допомогою IntelliTrace, **див. розділ Запис виконання коду за допомогою IntelliTrace для налаштування у Visual Studio за адресою: https://msdn.microsoft.com/ru-ru/library/dd572114(v=vs.110).aspx**.

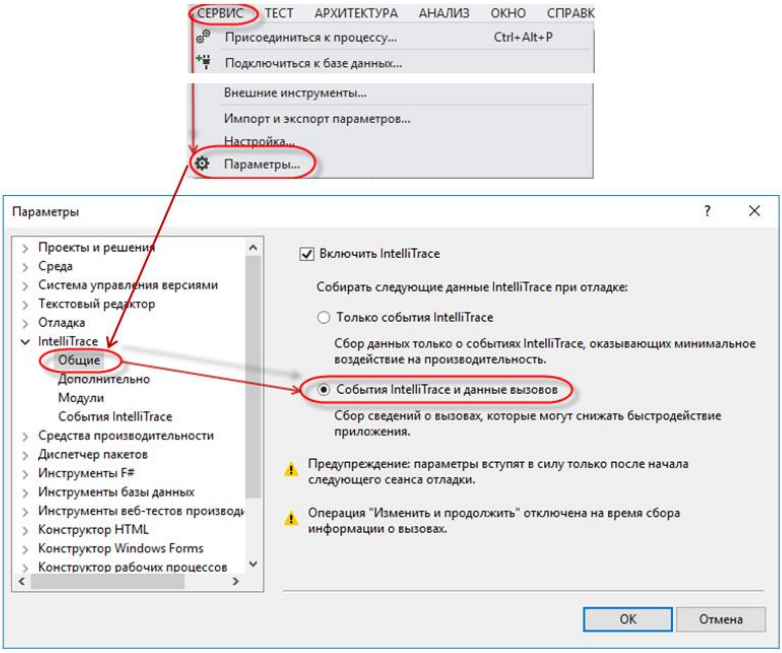


Рис. 3.32. Налаштування колекції викликів **IntelliTrace**

> **Увага:** включення збірки відомостей про виклики може значно уповільнити застосування. Воно також може збільшити розмір будь-яких файлів журналу **IntelliTrace** (файлів **.iTrace**), які зберігаються на диск. Щоб звести до мінімуму зазначені ефекти, можна збирати інформацію про виклики тільки у важливих для вас модулях. Щоб змінити максимальний розмір файлів **.iTrace**, перейдіть до **Сервис | Параметры | IntelliTrace | Дополнительно**.

Дані, які **IntelliTrace** збирає для функціональних викликів, наступні:

- Ім’я функції.

- Значення простих типів даних, переданих як параметри в точках входження функції і повернутих в точках виходу функції.
- Значення автоматичних властивостей, коли ті читаються або змінюються.
- Покажчики на дочірні об’єкти першого рівня, але не їх значення (тільки інформацію про те, чи є вони **null** чи ні).

3.5.1.3. Налаштування колекції модулів

Для того, щоб контролювати, як багато інформації про виклики збирає **IntelliTrace**, вкажіть тільки ті модулі, які вас цікавлять. Це може підвищити продуктивність застосування під час збору інформації. Щоб перевірити, що **IntelliTrace** збирає дані про виклики, перейдіть до **Сервис | Параметры | IntelliTrace | Общие**.

1. Вкажіть модулі для включення в колекцію і виключення з неї (див. **рис. 3.33**).
2. Щоб додати декілька модулів, треба використати підстановочний знак ***** на початку або кінці рядка. Для імен модуля використайте імена файлів або імена збірок. Шляхи до файлів не приймаються.

Щоб розпочати налаштування за допомогою IntelliTrace, **див. розділ Запис виконання коду за допомогою IntelliTrace для налаштування у Visual Studio за адресою: https://msdn.microsoft.com/ru-ru/library/dd572114(v=vs.110).aspx**.

3.6. Вікно Watch

Вікна **Watch (Контрольные значения)** призначені для спостереження за змінними. Доступитися до вікон **Watch** можна з меню або панелі інструментів **Debug**. Чотири вікна **Watch** називають **Watch 1, Watch 2, Watch 3** і **Watch 4**. Ці вікна **Watch** дозволяють налаштувати чотири списки елементів, за якими можна вести спостереження під час виконання програми. Використання окремих списків змінних зручне у разі, коли кожен список відноситься до різних зон видимості в коді застосування.

У вікно **Watch** можна додати змінну, щоб відстежувати, як змінюється її значення. Змінні не видаляються з вікна **Watch**, навіть

коли вони виходять із зони видимості. Так само як і у вікнах **Locals** та **Autos**, у вікні **Watch** відображаються стовпці **Name** (**Имя**), **Value** (**Значение**) і **Type** (**Тип**), із відомостями про змінні; тут можна змінювати поточні значення простих змінних вводячи нові значення в стовпець **Value**. Додати змінну у вікно **Watch** можна одним з наступних способів:

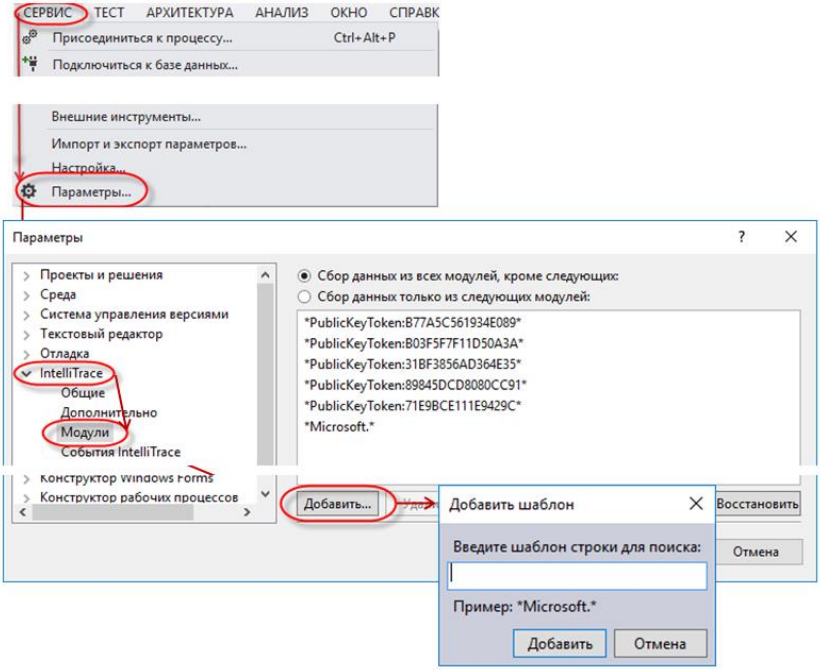


Рис. 3.33. Налаштування колекції модулів **IntelliTrace**

- двічі клацнути мишою порожній рядок у вікні **Watch** і ввести в нього ім'я змінної;
- у редакторі коду клацнути правою кнопкою миші змінну і з контекстного меню вибрати команду **Add Watch**;
- клацнути змінну в редакторі коду, вікнах **Autos** або **Locals** і перетягнути її у вікно **Watch**;
- перетягнути виділений елемент у вікно **Watch** за допомогою миші.

Середовище VS дозволяє відкривати декілька вікон **Watch**, що дає можливість відстежувати одночасно декілька наборів змінних. Вікно **Watch** представлене на **рис. 3.34**.

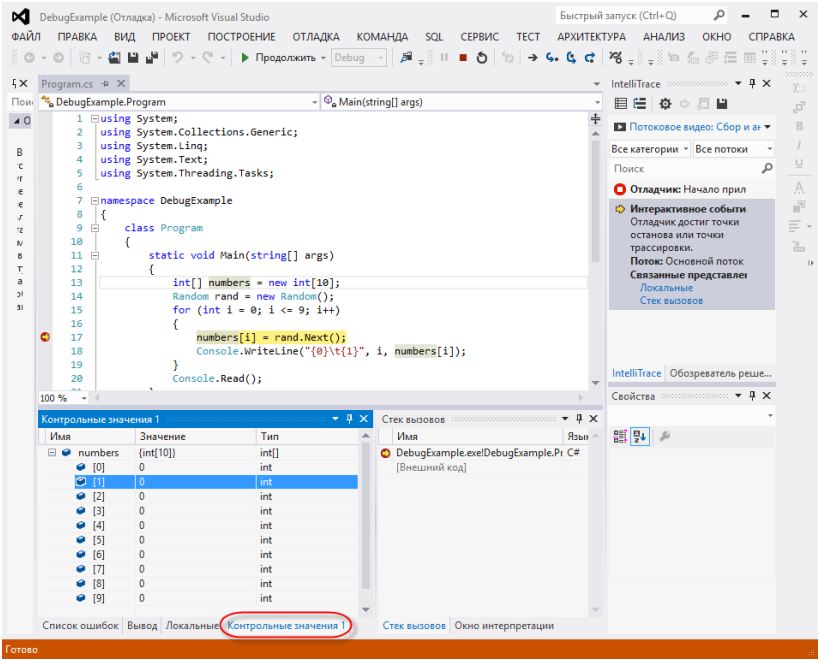


Рис. 3.34. Вікно **Watch**

3.7. Вікно QuickWatch

Діалогове вікно **QuickWatch (Быстрая проверка)** (**рис. 3.35**) призначене для швидкого розрахунку значення змінної. У стовпцях **Name**, **Value** і **Type** цього вікна відображуються відомості тільки про одну змінну; з нього змінну можна додати у вікно **Watch**, а також змінити її значення, ввівши нове значення в стовпець **Value**.

Для виклику діалогового вікна **QuickWatch** потрібно в редакторі коду клацнути правою кнопкою миші на змінній і вибрати з контекстного меню опцію **QuickWatch**.

З вікна **QuickWatch** можна також додати змінну або вираз у вікно **Watch**. Для цього в редакторі коду потрібно виділити

потрібний об’єкт, клацнути на ньому правою кнопкою миші і вибрати в контекстному меню пункт **Add Watch**. При цьому виділена змінна буде поміщена у вікно **Watch**.

У вікні **QuickWatch** можна писати вирази і додавати їх у вікно **Watch**. Доданий у вікно **QuickWatch** елемент буде обчислений при натисненні кнопки **Reevaluate**. Натиснення кнопки **Add Watch** відправить змінну до вікна **Watch 1**.

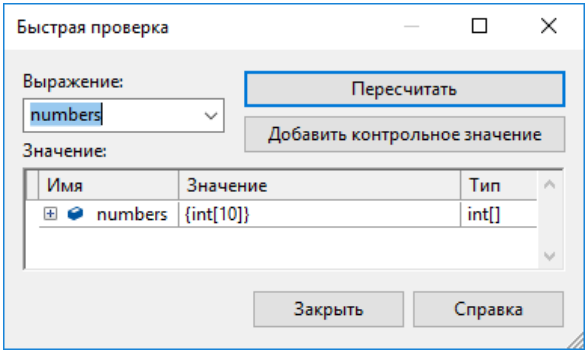


Рис. 3.35. Вікно **QuickWatch**

3.8. Вікно Command в режимі Immediate

Вікно **Immediate (Интерпретация/Непосредственное исполнение) (рис. 3.36)** – призначене для ручного введення і виконання команд. Це вікно з’являється автоматично при перериванні роботи програми в точках зупину програми. Для виконання команди або оператора треба написати команду і натиснути клавішу **<Enter>**.

Вікно **Command (Окно команд)** в режимі **Immediate (Интерпретация)** застосовують для виконання процедур при налаштуванні, розрахунку значень виразів або зміни значень змінних. Щоб відкрити вікно **Command** в режимі **Immediate**, клацніть команду **Debug | Windows | Immediate**.

Вікно **Command** в режимі **Immediate** дозволяє виконати будь-який допустимий оператор, але не дозволяє оголошувати змінні та об’єкти. У режимі покрокового виконання коду можна ввести у вікно **Command** оператор або ім’я методу так само, як в редакторі коду. Натиснувши клавішу **<Enter>**, ви змусите Visual Studio виконати

введений оператор. Завершивши виконання оператора або методу, застосування повернеться в режим покрокового виконання.

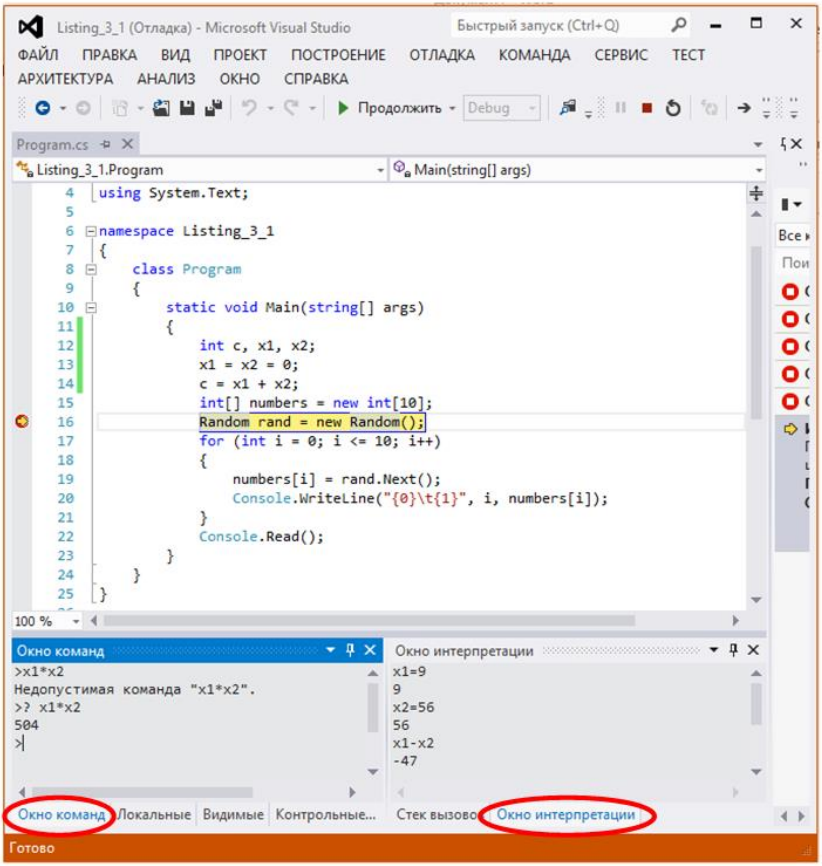


Рис. 3.36.Вікно Command в режимі **Immediate**

Коли застосування виконується покроково, значення виразів і змінних можна вводити у вікно **Command** безпосередньо, поставивши спереду знак питання. Наприклад, так можна перемножити поточні значення **x1** і **x2**, отримавши результат у вікні **Command (рис. 3.36)**:

? x1*x2

3.9. Спливаючі підказки даних DataTips

Спливаючі підказки даних **DataTips** дозволяють виділяти змінну або вираз в коді програми та отримати інформацію про значення змінної прямо у вікні редактора коду. Для отримання спливаючої підказки треба помістити покажчик миші над ім'ям відповідної змінної у вікні редактора коду. При цьому поряд з цією змінною з'являється невелике спливаюче віконце, в якому буде відображено значення цієї змінної і яке, крім того, може бути розширене для представлення детальнішої інформації.

Спливаючі підказки **DataTips** аналогічні дії вікна **QuickWatch**. Проте набагато простіше навести курсор на потрібну змінну, і її дані відобразяться в підказці **DataTip**. Приклад спливаючої підказки даних показаний на **рис. 3.37**.

Клацання правою кнопкою миші по знаку «*плюс»* (наразі, після клацання він перетворився на «*мінус»*) для цієї змінної розгортає множину членів цього об'єкта. Можна прокрутити цей список за допомогою стрілки в нижній частині вікна, якщо вміст списка виходить за межі вікна. Можна також клацнути правою кнопкою миші по будь-якому членові списка і відредагувати його значення, скопіювати його або додати у вікно спостереження.

3.10. Вікна візуалізації даних

Visual Studio пропонує швидкий і простий спосіб доступу до даних всередині об'єкта за допомогою вікон візуалізації даних. Ці вікна призначені для представлення даних об'єкта певним осмисленим чином.

Середовище розробки Visual Studio 2013 надає декілька типів вікон візуалізації даних:

- **HTML** – показує діалогове вікно у вигляді браузера, де HTML інтерпретований так, як його бачитиме користувач;
- **XML** – показує XML в структурованому форматі;
- **Text** – показує рядкове значення в легкому для читання форматі;
- **DataSet** – показує вміст об'єктів DataSet, DataView і DataTable.

Надалі при створенні і налаштуванні застосувань вам доведеться активно користуватися цими вікнами.

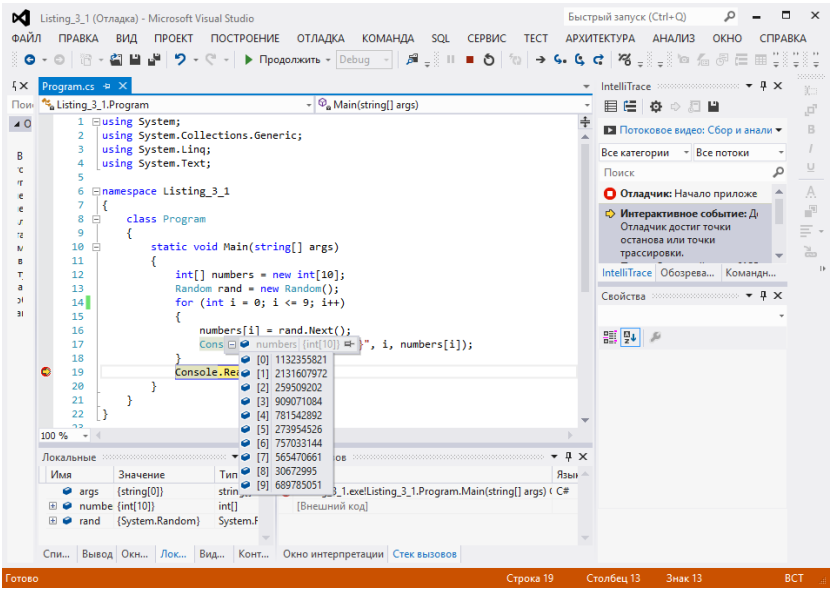


Рис. 3.37. Перегляд значення змінної у спливаючій підказці

3.11. Виключення

Обробка виключень в коді гарантує, що застосування зможе завжди успішно справлятися з можливими проблемами, проте в процесі налаштування коду застосування, автоматична обробка виключень, які виникають під час налаштування, часто є небажаною, особливо в тих випадках, коли обробка виключень передбачає скидання даних і припинення виконання застосування. Однак при налаштуванні застосування вимагається, щоб відлагоджувач допоміг встановити причину генерації виключення.

При виникненні виключення середовище часу виконання намагається знайти відповідний обробник, навіть у тому випадку, якщо в програмному коді він не передбачений. Коли з'ясується, що такий обробник відсутній, виконання вашої програми буде припинено. Генерація виключення супроводжується очищенням

стека викликів, у результаті стає неможливим перегляд значення змінних, бо всі вони покинуть межі зони видимості.

Щоб змінювати поведінку програми при генерації виключення, Visual Studio надає вікно **Exceptions… (Исключения…)**, яке можна відкрити в меню **Debug | Exceptions… (Отладка | Исключения…)** (**рис. 3.38**), яке дозволяє вказати, що повинно відбуватися при виникненні виключення. Доступні два можливі варіанти вибору:

- продовжити виконання програми;
- припинити виконання програми (у цьому випадку відлагоджувач починає самостійно виконувати оператор **throw**).

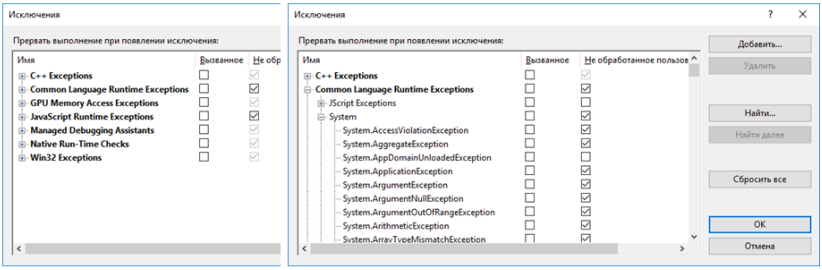


Рис. 3.38. Вікно **Exceptions**

Середовище VS не в змозі автоматично відстежувати написані вами класи виключень користувача, але ви можете вручну додати їх у список і тим самим вказати, які з них повинні приводити до негайного припинення виконання застосування. Для додавання класів виключень користувача потрібно клацнути по кнопці **Add (Добавить)** і ввести ім’я потрібного класу виключень.

3.12. Класи Debug і Trace

Класи **Debug** і **Trace** дозволяють генерувати і записувати в журнали повідомлення з інформацією про стан застосування, не перериваючи його виконання. Клас **Debug** дозволяє генерувати і записувати в журнал під час виконання повідомлення з описом стану програми, а клас **Trace** – створювати інструменти для діагностики навіть для скомпільованих застосувань.

3.12.1. Виведення трасування у вікно Output

Класи **Trace** і **Debug** містять статичні методи, які дозволяють перевіряти умови під час виконання і записувати результати в журнал. Дані, згенеровані ними, відображуються у вікні **Output** (**рис. 3.39**), де їх можна переглядати при налаштуванні, і записувати в колекцію **Listeners**. Колекція **Listeners** містить групу класів, які отримують дані від **Trace** і **Debug**, їх називають *слухачами*. Класи **Trace** і **Debug** використовують спільну колекцію **Listeners**, надаючи одні і ті ж дані всім *слухачам* з цього набору.

Різні види слухачів записують ці текстові файли або журнали подій. Після завершення виконання програми дані трасувань, записані слухачами, аналізують, щоб знайти помилки в застосуванні. Трасування корисне і при налаштуванні програм. Приклад програми з використанням трасування наведений в **лістингу 3.2**.

```
// Лістинг 3.2. Програма з виведенням трасувальних повідомлень
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.IO;

namespace Listing_3_2
{
    class Program
    {
        static void Main(string[] args)
        {
                int[] numbers = new int[10];
                Random rand = new Random();
                for (int i = 0; i < 10; i++)
                {
                    numbers[i] = rand.Next();
                    Console.WriteLine("{0}\t{1}", i, numbers[i]);
                    Trace.WriteLine(String.Format("{0}:\t{1}\t{2}",
```

```
                    DateTime.Now, i, numbers[i]));
                }
                Trace.Flush();
                Console.Read();
            }
        }
}
```

Класи **Trace** і **Debug** надають однакові набори методів і властивостей та записують результати в колекцію **Listeners**. Єдина відмінність між цими класами в тому, що за замовчуванням оператори **Debug** видаляються при компіляції застосування в **Release**-версію, а оператори **Trace** там залишаються. Тому клас **Debug** зазвичай застосовують для налаштування на стадії розробки, а **Trace** – для тестування і налаштування після компіляції остаточної версії застосування.

Компілятор ігнорує виклики **Debug** і **Trace**, якщо вони не визначені як символ умовної компіляції. Прапорці для визначення **Debug** і **Trace** розташовані у вікні властивостей проекту на вкладці **Build** (**рис. 3.40**), їх можна включати або відключати залежно від того, чи потрібно здійснювати запис інформації трасування, або налагоджування.

3.12.2. Запис даних в набір Listeners

Усі дані, згенеровані класами **Debug** і **Trace**, спрямовуються в набір **Listeners**. Це спеціальний набір, який упорядковує класи, здатні отримувати дані від **Trace**, і який надає доступ до цих класів. Кожен член набору **Listeners** в повному обсязі отримує дані, які генеруються класами **Debug** і **Trace**, спосіб обробки цих даних залежить від слухача.

Дані, згенеровані операторами **Debug** і **Trace**, передаються членам набору **Trace.Listeners** – об’єктам, здатним отримувати дані трасувань і обробляти їх. Існує три види слухачів трасувань:

- DefaultTraceListener;
- TextWriterTraceListener;
- EventLogTraceListener.

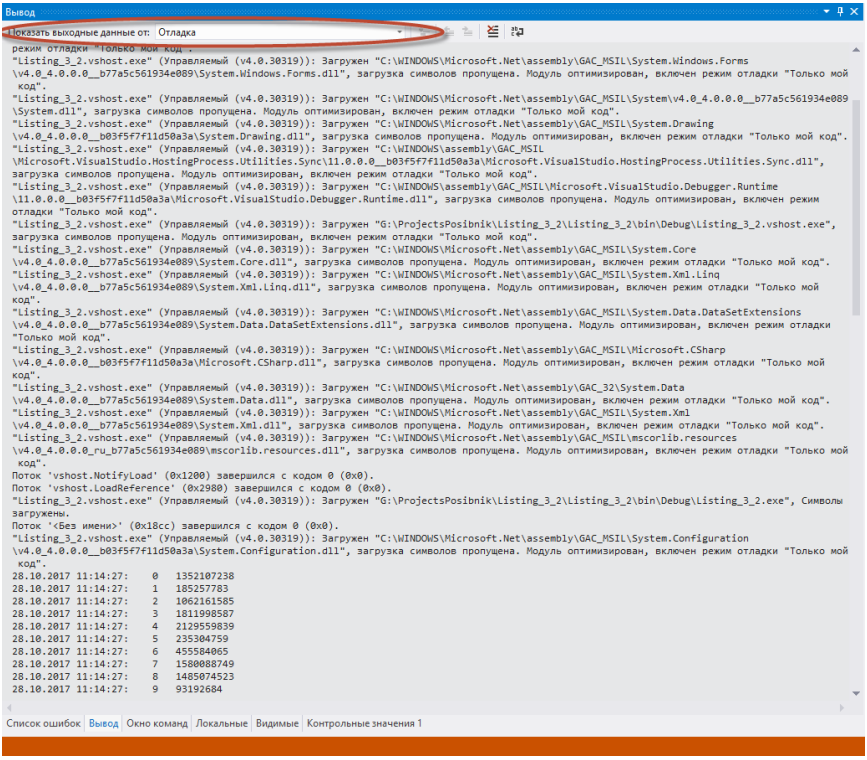


Рис. 3.39. Дані трасування у вікні **Output**

За замовчуванням набір **Listeners** ініціалізується єдиним членом, екземпляром класу **DefaultTraceListener**. Він створюється автоматично та отримує дані від **Trace** і **Debug** навіть у відсутність інших слухачів, доданих явно. Отримані дані **DefaultTraceListener** передає відлагоджувачу і відображує у вікні **Output** середовища розробки VS (як у попередньому розділі). Щоб створити журнал повідомлень трасувань, з яким можна працювати незалежно від відлагоджувача, потрібно додати принаймні один об’єкт **Listener**.

Для запису згенерованих класами **Trace** і **Debug** даних в набір **Listeners** служать методи цих класів:

- **Write()** – записує рядок в набір **Listeners**;

- **WriteLine()** – записує в набір **Listeners** рядок, завершуючи його символом повернення каретки;
- **WriteIf()** – записує рядок в набір **Listeners**, якщо заданий булевий вираз має значення **true**;
- **WriteLineIf()** – записує в набір **Listeners** рядок, завершений символом повернення каретки, якщо заданий булевий вираз має значення **true**;
- **Assert()** – записує повідомлення в набір **Listeners**, якщо заданий булевий вираз має значення **false**, і відкриває вікно з текстом цього повідомлення;
- **Fail()** – створює перевірку, яка автоматично завершується невдачею без перевірки умови. При цьому в набір **Listeners** записується повідомлення і відображується вікно з текстом цього повідомлення.

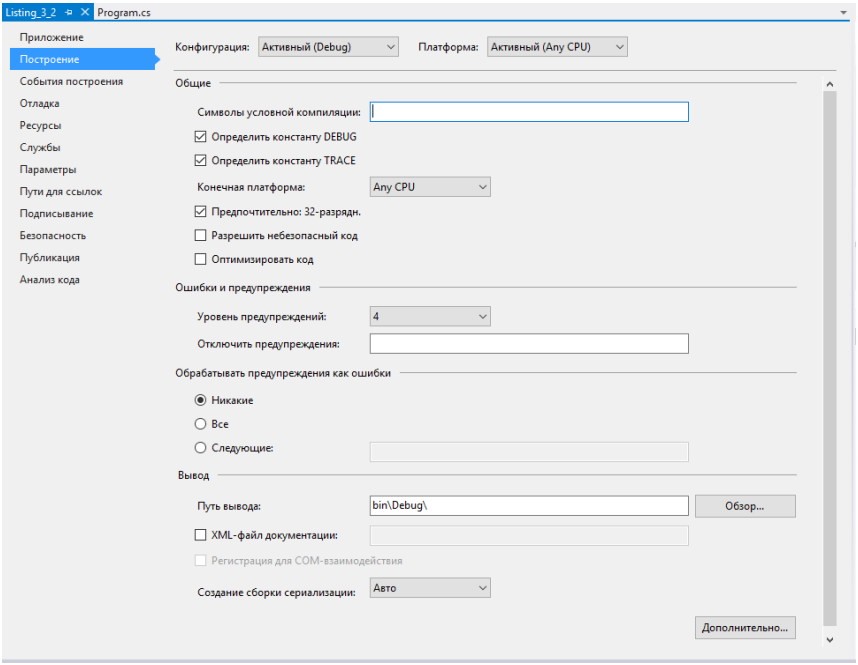


Рис. 3.40. Встановлення прапорців **Debug** і **Trace** в опціях компілятора

Повідомлення трасувань можна форматувати, змінюючи відступи в тексті, викликаючи методи **Indent()** та **Unindent()** і встановлюючи властивості **IndentSize** та **IndentLevel**. Ці методи і властивості дозволяють створити ієрархію повідомлень про помилки. Властивість **IndentSize** повертає або встановлює число пропусків, яке складає один рівень відступу, а **IndentLevel** – рівень відступу для поточного повідомлення. Метод **Indent** збільшує, а **Unindent** – зменшує значення **IndentLevel** на одиницю.

Щоб записати дані в набір **Listeners**, потрібно викликати відповідний метод:

- **Write()** або **WriteLine()** – для безумовного запису;
- **WriteIf()** або **WriteLineIf()** – щоб записати дані, якщо перевірка умови дає **true**;
- **Fail()** – для безумовного запису результату і відображення повідомлення;
- **Assert()** – щоб записати дані, якщо перевірка умови дає **true** і відобразити повідомлення.

Дані, згенеровані класом **TextWriterTraceListener**, записуються в текстовий файл, в об'єкт **Stream** або в об'єкт **TextWriter**. Об'єкти **Stream** і **TextWriter** також застосовуються для запису даних в текстові файли. Щоб створити об'єкт **TextWriterTraceListener** для запису даних, згенерованих класом **Trace**, в текстовий файл, потрібно створити або відкрити файл, куди будуть записуватися дані, після цього потрібно створити екземпляр **TextWriterTraceListener**, передавши йому створений відкритий файл. На завершення потрібно додати **TextWriterTraceListener.Listeners**.

Приклад запису подій трасувань у файл показаний в **лістингу 3.3**.

```
// Лістинг 3.3. Запис подій трасування у файл
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
using System.Diagnostics;
using System.IO;

namespace Listing_3_3
{
class Program
 {
      static void Main(string[] args)
      {
      // Створюємо екземпляр об'єкта FileStream, який вказує на потрібний
      текстовий файл
           FileStream traceLog = new FileStream("G:\\TraceLog.txt",
                     FileMode.OpenOrCreate);
      // Створюємо екземпляр TextWriterTraceListener, передавши йому новий
      об'єкт FileStream
           TextWriterTraceListener textListener = new
                     TextWriterTraceListener(traceLog);
      // Додамо створений об'єкт Listener до набору Listeners.
           Trace.Listeners.Add(textListener);
           int[] numbers = new int[10];
           Random rand = new Random();
           for (int i = 0; i < 10; i++)
           {
                     numbers[i] = rand.Next();
                     Console.WriteLine("{0}\t{1}", i, numbers[i]);
                     textListener.WriteLine(String.Format("{0}:\t{1}\t{2}",
                               DateTime.Now, i, numbers[i]));
           }
           // Скинути буфер на диск
           Trace.Flush();
           traceLog.Close();
      }
}
}
```

Якщо відкрити існуючий файл за допомогою **FileMode.OpenOrCreate**, вміст файла буде записаний повторно. Щоб додати нові дані до існуючого файла, оголошуйте об'єкт

FileStream за допомогою **FileMode.Append**. Приклад записаних у файл даних трасувань представлений на **рис. 3.41**.

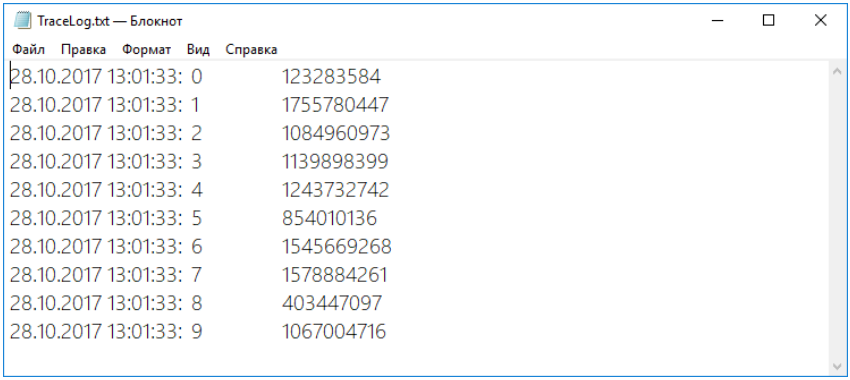


Рис. 3.41. Текстовий файл з даними трасування

Для запису даних трасувань в об'єкт **EventLog** застосовують об'єкт **EventLogTraceListener**. У принципі, запис даних трасувань в **EventLog** не відрізняється від запису в текстовий файл. Потрібно створити об'єкт **EventLogTraceListener**, передавши йому новий об'єкт **EventLog** (його треба створити наперед) або посилання на існуючий об'єкт журналу подій, і додати його до набору **Listeners**. Після цього результати **Trace** записуватимуться у вказаний об'єкт **EventLog** у вигляді об'єктів **EventLogEntry**.

Приклад програми, яка використовує запис даних трасувань в журнал подій, представлений в **лістингу 3.4**.

```
// Лістинг 3.4. Запис даних трасування в журнал подій
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.IO;
```

```
namespace Listing_3_4
{
    class Program
    {
        static void Main(string[] args)
        {
                int[] numbers = new int[10];
                Random rand = new Random();
                // Створимо екземпляр журнала подій EventLog
                EventLog log = new EventLog("ConsoleApp Debug Log");
                        log.Source = "Trace Output";
                // Створюємо об'єкт EventLogTraceListener і передаємо йому
                об'єкт EventLog
                EventLogTraceListener eventlogListener =
                        new EventLogTraceListener(log);
                for (int i = 0; i < 10; i++)
                {
                        numbers[i] = rand.Next();
                        Console.WriteLine("{0}\t{1}", i, numbers[i]);
                        eventlogListener.WriteLine(String.Format("{0}\t{1}", i,
                         numbers[i]));
                }
                Trace.Flush();
                Console.Read();
        }
    }
}
```

Якщо виконати програму, а потім відкрити **Event Viewer (Просмотр событий)**, то в директорії **Application and Services Logs (Журнал приложений и служб)** ми побачимо новий журнал подій **ConsoleApp Debug Log** для нашого застосування, показаний на **рис. 3.42**.

Для успішного трасування журналів треба запускати Visual Studio з правами адміністратора, інакше доступ до перегляду подій буде заборонений.

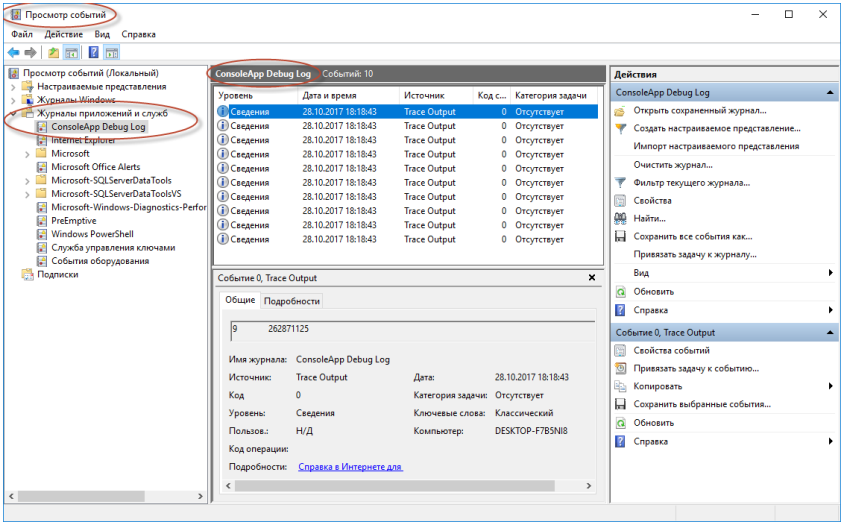


Рис. 3.42. Запис даних трасувань в журнал подій

Примітка:

Перегляд подій – це засіб, який відображує докладні відомості про важливі події на комп'ютері (наприклад, програми, які не були запущені за розкладом, або оновлення, які було автоматично завантажено). Він може бути корисним під час усунення неполадок і помилок із системою Windows та іншими програмами.

Засіб **«Перегляд подій»** записує відомості в кілька різних журналів. Журнали Windows включають:

- **Події застосувань (програм).** Події поділяються на *помилки*, *попередження* та *повідомлення*, залежно від серйозності події. *Помилка* – це значна проблема, наприклад, втрата даних. *Попередження* – це подія, яка необов'язково є значною, але може вказувати на можливу проблему в майбутньому. Інформаційна подія описує успішну операцію програми, драйвера або служби.
- **Події, пов'язані з безпекою.** Ці події називаються *аудитами* й описуються як *успішні* або *невдалі*, залежно від події, наприклад, чи успішний був вхід користувача в систему Windows.

- **Події налаштовування.** Для комп'ютерів, налаштованих як контролери домену, тут відображатимуться додаткові журнали.
- **Системні події.** Системні події реєструються системою Windows і системними службами Windows і класифікуються як *помилки, попередження* або *повідомлення*.
- **Переадресовані події.** Це події, переадресовані до журналу іншими комп'ютерами.

Журнали застосувань і служб різняться. Вони включають окремі журнали про запущені на комп'ютері програми, а також більш деталізовані журнали, які стосуються окремих служб Windows.

Відкриття засобу «Просмотр событий». Для цього натисніть кнопку **Пуск** і послідовно клацніть **Служебные-Windows | Панель управления | Система и безопасность | Администрирование | Просмотр журналов событий**. Якщо буде запропоновано ввести пароль адміністратора або підтвердити видалення, введіть пароль, або надайте підтвердження. Цей сценарій для нашого комп'ютера; у вас може бути інша послідовність кроків сценарію.

3.12.3. Перемикачі трасувань

Дані, згенеровані операторами **Debug** і **Trace**, інтенсивно використовуються при налаштуванні, але після компіляції і випуску остаточної версії застосування трасування включають тільки в особливих випадках.

Бібліотека базових класів **.NET Framework** містить два типи трасувальних перемикачів. *Перший*, **BooleanSwitch**, повертає значення типу **Boolean**, що робить його схожим на звичайний перемикач, який не має проміжних положень. *Другий*, **TraceSwitch** підтримує параметр з п'ятьма можливими значеннями, що дозволяє встановити для перемикача потрібний тип результату.

Включати і вимикати відображення даних, згенерованих операторами **Trace**, можна засобами перемикачів трасувань, якими управляють через конфігураційний файл застосування.

Перемикачі трасувань дозволено включати і відключати у скомпільованому застосуванні через його конфігураційний файл – файл з розширенням **config** у форматі **XML**, в якому зберігається

потрібна застосуванню інформація. Конфігураційний файл розташовується в одному каталозі з виконуваним файлом.

Конфігураційні файли є не у всіх застосувань. Якщо у вашого застосування, яке використовує перемикачі трасувань, такого файла немає, його потрібно створити.

При створенні перемикача трасування треба створити параметр **DisplayName**, який визначає ім'я перемикача в конфігураційному файлі. При редагуванні конфігураційного файла треба вказати ім'я перемикача і цілочисельне значення, яке потрібне йому присвоїти. Для об'єктів типу **BooleanSwitch** значення **0** деактивує перемикач, а будь-які значення, відмінні від нуля, активують його.

Клас **TraceSwitch** має властивість **TraceLevel**, значення якої представляють різні рівні помилок. Ця властивість здатна приймати будь-яке з п'яти значень перерахування **TraceLevel**:

- **TraceLevel.Off(0)** – деактивує об'єкт **TraceSwich**;
- **TraceLevel.Error(1)** – визначає виведення коротких повідомлень про помилки;
- **TraceLevel.Warning(2)** – дозволяє виводити повідомлення про помилки і попередження;
- **TraceLevel.Info(3)** – дозволяє виводити повідомлення про помилки, попередження і короткі інформаційні повідомлення;
- **TraceLevel.Verbose(4)** – дозволяє виводити повідомлення про помилки, попередження і детальні відомості про виконання програми.

Щоб створити конфігураційний файл і налаштувати перемикачі трасування, потрібно спочатку створити в програмі об'єкти перемикачів з відповідними значеннями властивості **DisplayName**. Потім, якщо у вашого застосування немає файла конфігурації (з розширенням .**config**), потрібно його створити. Для цього виберіть в меню **Project** опцію **Add New Item**, у діалоговому вікні **Add New Item (Добавление нового элемента)**, яке відкрилося, виберіть **Application Configuration File (Файл конфигурации приложения)** і назвіть новий файл **App1.config**, як показано на **рис. 3.43**.

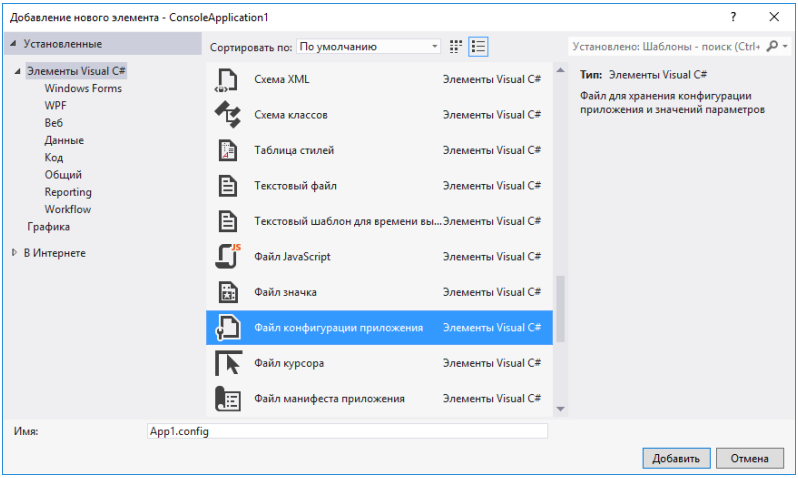


Рис. 3.43. Додавання файла конфігурації в застосування

Між тегами **<configuration>** і **</configuration>** додайте **XML**-код, який оголошує перемикачі трасувань **IsTraceOn** (типу **BooleanSwitch**) і **SwitchLevel** (типу **TraceSwitch**) і присвоює їм значення, як показано в **лістингу 3.5**.

Лістинг 3.5. Перемикачі трасувань

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.diagnostics>
        <switches>
                <add name="IsTraceOn" value="1" />
                <add name="SwitchLevel" value="3" />
        </switches>
    </system.diagnostics>
</configuration>
```

У коді **лістингу 3.5** значення **IsTraceOn** встановлене в **true**, а значення **SwitchLevel** – в **TraceLevel.Info**. Щоб включити або вимкнути будь-який з перемикачів, досить буде змінити його значення в конфігураційному файлі, не здійснюючи повторну компіляцію застосування.

Екземпляри класів **BooleanSwitch** і **TraceSwitch** створюються так само, як екземпляри будь-якого іншого класу. Конструктори цих типів перемикачів вимагають два параметри: **DisplayName**, який визначає ім'я перемикача, яке відображується в інтерфейсі користувача, і **Description**, який задає короткий опис перемикача, наприклад:

```
BooleanSwitch bSwitch = new BooleanSwitch("IsTraceOn", "Tracing");
TraceSwitch trSwitch = new TraceSwitch("SwitchLevel", "Tracing");
```

За допомогою перемикачів трасувань оператори **Trace** перевіряють, чи потрібно записувати дані трасувань. Методи **Trace.WriteIf** і **Trace.WriteLineIf** дозволяють перевірити, чи активний перемикач трасування, і розпізнати визначуваний ним рівень трасування:

```
Trace.WriteLineIf(bSwitch.Enabled == true, "Trace Boolean switch");
Trace.WriteLineIf(trSwitch.Level == TraceLevel.Info, "Trace level switch");
```

Оператори **Trace** не підключаються до перемикачів трасувань автоматично, це потрібно написати програмістові в коді програми. Під'єднання перемикачів трасування у програмі показане в **лістингу 3.6**.

```
// Лістинг 3.6. Під'єднання перемикачів трасування в програмі
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;

namespace Listing_3_6
{
class Program
{
    static void Main(string[] args)
    {
```

```
            int[] numbers = new int[10];
            Random rand = new Random();
            BooleanSwitch bSwitch = new BooleanSwitch("IsTraceOn", "Tracing");
            TraceSwitch trSwitch = new TraceSwitch("SwitchLevel", "Tracing");
            Trace.WriteLineIf(bSwitch.Enabled == true, "Trace Boolean switch:");
            for (int i = 0; i < 10; i++)
            {
                        numbers[i] = rand.Next();
                        Console.WriteLine("{0}\t{1}", i, numbers[i]);
                        Trace.WriteLineIf(bSwitch.Enabled == true, String.Format(
                                    "{0}:\t{1}\t{2}", DateTime.Now, i, numbers[i]));
            }
            Trace.WriteLineIf(trSwitch.Level == TraceLevel.Info, "Trace level
            switch:");
            for (int i = 0; i < 10; i++)
            {
                        numbers[i] = rand.Next();
                        Console.WriteLine("{0}\t{1}", i, numbers[i]);
                        Trace.WriteLineIf(trSwitch.Level == TraceLevel.Info,
                                    String.Format("{0}:\t{1}\t{2}",
                                    DateTime.Now, i, numbers[i]));
            }
            Trace.Flush();
            Console.Read();
      }
}
}
```

При виконанні коду, який створює перемикач трасування, застосування перевіряє наявність у файлі конфігурації відомостей про цей перемикач. Для кожного перемикача перевірка здійснюється один раз при запуску застосування. Щоб змінити параметри перемикача трасування після його створення, треба зупинити застосування, внести відповідні зміни до файла конфігурації і запустити його знову.

Примітка: При використанні методів, які виконують безумовний запис, наприклад **Trace.Write()** або

Trace.WriteLine(), результати трасування записуються в усіх випадках, незалежно від стану перемикачів.

3.13. Приклад налаштування за допомогою Visual Studio

Тут ми використаємо наступні ключові концепції:

- наявний інструментарій для налаштування;
- встановлення точок зупину;
- дослідження статусу програми;
- рішення проблем за допомогою налагоджувальних засобів VS.

Наш код часто містить помилки – частіше, ніж нам хотілося б. На щастя, на випадок виникнення потреби в усуненні помилок VS пропонує широкий набір налагоджувальних інструментів. Продемонструємо, як використовувати налагоджувач VS для усунення проблем в роботі коду. Покажемо, як вирішувати проблеми шляхом встановлення точок зупину, шляхом покрокового виконання програми і дослідження статусу програми. Крім того, використаємо інструментарій, призначений для дослідження структури коду програми під час розробки. Окрім встановлення точок зупину також, виконаємо їх індивідуальне налаштування і будемо управляти списком точок зупину. Також використаємо опції, доступні при покроковому виконанні коду. Крім того, практично використаємо способи, застосовуючи які можна визначати значення різних змінних, а також використаємо різні інструменти для вивчення коду програми. Почнемо з обговорення фрагмента коду, на прикладі якого демонструватимуться концепції налаштування, викладені тут.

3.13.1. Код, для демонстрування прийомів налаштування

Щоб продемонструвати налаштування повнофункціонального застосування, яке використовується в реальному житті, потрібно буде написати багато сторінок коду. Для посібника це – забагато і крім того, такий підхід серйозно ускладнить розуміння основ. Тому ми розглянемо приклад, який імітує повномасштабне застосування. Хоча це – всього лише модель, для нас цього прикладу буде цілком достатньо. При налаштуванні вам треба буде проходити ієрархічними деревами у складі вашого коду, де один метод викликає

інший, який, залежно від виконуваних завдань вашою програмою, може мати декілька рівнів вкладеності. Код, в прикладах, які будуть демонструватися, матиме багато рівнів викликів, так що це наочно проілюструє техніку налаштування програм у VS.

У **лістингу 3.7** показано приклад використовуваного нами коду. Програма є консольним застосуванням.

Створимо новий проект консольного застосування за відомою вам схемою. Застосування, код якого приведений у **лістингу 3.7**, обчислює знижку для клієнта, базуючись на процентній ставці, встановленій для цього клієнта, і на тому товарі або послузі, які були цим клієнтом замовлені.

```
// Лістинг 3.7. Код, призначений для ілюстрації
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// C#: Program.cs
namespace DebugAndTestDemo
{
    class Program
    {
        static void Main(string[] args)
        {
                Customer cust = new Customer();
                cust.Discount = .1m;
                Order ord = new Order();
                ord.AddItem(5.00m);
                ord.AddItem(2.50m);
                cust.Order = ord;
                decimal discount = cust.GetOrderDiscount();
                Console.WriteLine("Customer Discount: {0}", discount);
                Console.ReadKey();
        }
    }
}
```

```
// C#: Customer.cs
namespace DebugAndTestDemo
{
	class Customer
	{
		public decimal Discount { get; set; }
		public Order Order { get; set; }
		public decimal GetOrderDiscount()
		{
			return Order.Total * Discount;
		}
	}
}

// C#: Order.cs
//using System.Collections.Generic;
namespace DebugAndTestDemo
{
	class Order
	{
		private List <decimal> orderItems = new List <decimal>();
		public decimal Total
		{
			get
			{
				decimal amount = 0;
				foreach (var item in orderItems)
				{
					amount = amount + item;
				}
				return amount;
			}
		}
		public void AddItem(decimal amount)
		{
			orderItems.Add(amount);
		}
```

```
    }
}
```

Навіть побіжний погляд на код, приведений у **лістингу 3.7**, показує, що ця програма складніша, ніж приклад, який розглядався у попередніх розділах. Щоб зрозуміти, що відбувається при запуску програми, почнемо її розгляд з методу **Main**, який є *точкою входження застосування*. У методі **Main** створюються два екземпляри об'єктів – **Customer (Клієнт)** та **Order (Замовлення)**.

Після того, як буде створений екземпляр об'єкта **Customer**, можна переглянути його властивість **Discount (Знижка)**, яка набуває значення **.1 (10%)**. Це означає, що кожний клієнт може отримати індивідуальну знижку, що може бути корисним, якщо ви бажаєте, наприклад, заохочувати постійних клієнтів, які купляють у вас ті або інші товари.

Далі, створюється екземпляр об'єкта **Order (Замовлення)**, після чого відбувається виклик методу **AddItem**, який додає елемент до посилання на замовлення **ord**. Цей код тільки додає обсяг замовлення, а в реальній ситуації, швидше за все, в програмі повинне бути ще посилання на детальну інформацію про те, що саме було замовлено. Клас **Customer** має властивість **Order**, яку код передає й екземпляру **ord** класу **Order**. Після цього ви отримуєте конкретного клієнта (**Customer**) з величиною знижки, і цей екземпляр посилається на конкретне замовлення (**Order**), яке, у свою чергу, має елементи (тут вони представлені грошовою сумою – тільки з міркувань стислості).

Ця програма обчислює загальну знижку в грошовому виразі для цього клієнта, викликаючи метод **GetOrderDiscount** для екземпляра **Customer**, який повертає обчислену суму знижки. Потім це повернене значення виводиться на консоль. У принципі, в коді цієї програми була створена парочка екземплярів об'єктів, **cust** та **ord**, яким були передані потрібні їм дані, а також інструкції з їх обробки. У результаті роботи, виконаної кодом цієї програми, на консоль виводиться знижка в грошовому виразі, обчисленої для вказаного клієнта на підставі зробленого їм замовлення.

Увесь код, який міститься в методі **Main**, знаходиться на *першому рівні ієрархії* викликів. Методи і властивості, які

знаходяться в класах **Customer** та **Order**, розміщені на *другому рівні ієрархії*.

Уважно вивчивши код класу **Order**, ви неодмінно звернете увагу на те, що там є властивість **Total** і метод **AddItem**.

Метод **AddItem** додає параметр **item** у свою колекцію **orderItems**. Далі по колекції **orderItems** здійснюються ітерації з використанням властивості **Total** як змінної циклу, при цьому спочатку обчислюється сума повернення грошей по усіх замовлених елементах. Зверніть увагу, що клас **Customer** має властивість **Discount**, яка містить значення типу **decimal**, яке буде використовуватися як відсоток знижки. Метод **GetOrderDiscount** з класу **Customer** множить значення **Discount** на значення **Total** з класу **Order**, і повертає суму знижки, яка нараховується на це замовлення.

На цьому етапі дуже важливо ретельно вивчити приведений код і зрозуміти взаємовідносини та комунікації між різними об'єктами. Зверніть увагу, що *кожен клас має чітко визначену мету, причому його мета, зазвичай, чітко взаємозв'язана з його назвою*. Цей зв'язок (смислове навантаження) допомагає вирішити, які дані і які методи повинен мати клас. Наприклад, клас **Order** має властивість **Total** і метод **AddItem**, а клас **Customer** має властивість **Discount** і метод **GetOrderDiscount**. Кожен об'єкт підтримує комунікації з іншими об'єктами, і їх спільна робота служить для виконання загальної задачі. Наприклад, клас **Customer** повинен обчислювати знижку, тому що саме цей клас має відомості про те, якою має бути ця знижка (а інформація про це передана йому з методу **Main**). При цьому клас **Customer** повинен взаємодіяти з класом **Order**, тому що клас **Order** – це єдиний об'єкт, який володіє інформацією про те, що саме замовлено, в якому обсязі, і як обчислювати загальну суму замовлення в грошовому виразі.

Вам треба запустити програму і досліджувати внутрішню логіку її роботи. У VS є багато різних засобів, які дозволяють візуалізувати логіку програми і виконати її налаштування. Усі ці засоби будуть детальніше розглянуті далі.

3.13.2. Інструменти кодування, які спрощують розробку

Одним з інструментів VS 2013 є **Call Hierarchy (Иерархия вызовов)** – цей засіб дозволяє визначити, який код викликає той або

інший метод і які саме методи викликає ваш код. Спершу, давайте обговоримо, чому цей інструмент такий важливий, а потім проілюструємо його користь на конкретному прикладі. Вікно **Call Hierarchy** показане на **рис. 3.44**.

Ієрархія викликів повідомляє про ваш код великий обсяг цінної інформації, у тому числі, *міра повторного використання* (**degree of reuse**), *вплив змін*, а також *потенційну важливість процедури*. *Викликаюча точка* (**call site**) є кодом, який ініціює інші члени класу. Наприклад, в **лістингу 3.7**, метод **Main** є точкою виклику для методу **GetOrderDiscount**, який є кодом, що викликається.

З погляду повторюваності використання, *велике* число точок виклику у методу може вказувати на його відносну універсальність і можливість багатократного використання. З іншого боку, *невелике* число викликів на можливість повторного використання коду не вказує, а *нульове* число викликів говорить про те, що код практично не використовується і, ймовірно, може бути навіть видалений.

Велике число точок виклику може також говорити про те, що внесення змін в цей метод здійснить істотний вплив на роботу програми. Перегляд точок виклику, з яких надходять запити до методу, також може дати багато корисної інформації про те, куди і звідки передаються ті або інші значення, і які зміни можуть знадобитися у викликаючих методах (іншими словами, якщо деяка зміна у викликаючому методі потрібна, то який ланцюжок змін в методах, які викликаються, вона спричинить).

Все сказане демонструє важливість та актуальність дослідження ієрархії викликів. Тепер же давайте зосередимося на тому, як вона працює. По-перше, важливо зрозуміти, що ієрархія викликів є контекстно-чутливою. Це значить, код, який має фокус введення у вікні редактора, визначає і вашу точку зору. У прикладі, який розглядається нами, фокус має метод **GetOrderDiscount** з класу **Customer** і нам потрібно переглянути, звідки викликається метод **GetOrderDiscount,** і які фрагменти коду в його складі є точками виклику (**call sites**). Щоб досліджувати ієрархію викликів, або клацніть правою кнопкою миші по методу **GetOrderDiscount** та методу **AddItem** у вікні редактора та із контекстного меню, яке розкрилося виберіть команду **View Call Hierarchy**, або виділіть метод **GetOrderDiscount** у вікні редактора і натисніть клавіатурну

комбінацію **<CTRL>+<K>, <T>**. Visual Studio відобразить вікно **Call Hierarchy**, показане на **рис. 3.44**.

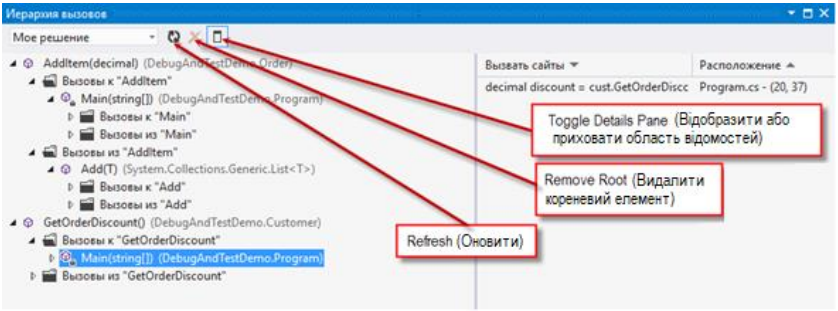


Рис. 3.44. Вікно **Call Hierarchy (Иерархия вызовов)**

Вікно **Call Hierarchy**, показане на **рис. 3.44**, демонструє виклики, які *надходять до методу* **GetOrderDiscount** (**гілка Calls To (Вызовы к)**), і виклики, які *виходять з цього методу* (**гілка Calls From (Вызовы из)**). Гілка **Calls To** є списком точок виклику до методу **GetOrderDiscount**. Список **Calls From** – це список тверджень, які входять до складу методу **GetOrderDiscount**, з яких здійснюються виклики до інших членів класу.

Список, який розкривається, у верхній частині вікна, показаного на **рис. 3.44**, в якому вибраний елемент **My Solution**, показує, як ієрархія викликів буде виконувати пошук, щоб знайти точки **Calls To** і **Calls From**. Для вибору доступні наступні опції: **My Solution, Current Project** і **Current Document** – їх зміст зрозумілий.

Якщо в процесі роботи над кодом вам захочеться відновити вікно **Call Hierarchy**, натисніть кнопку **Refresh (Обновить)**. Кожного разу, коли ви будете переглядати ієрархію викликів, вибраний елемент додаватиметься в список. Можна скористатися кнопкою **Remove Root (Удалить корневой элемент)**, щоб видалити елемент списка. Кнопка **Toggle Details Pane (Отобразить или скрыть область сведений)** дозволяє відображати або, навпаки, приховувати панель **Details**, на якій показано розташування точки виклику та її код. На **рис. 3.44** вибраний метод **Main**, внаслідок чого відображається виклик до методу **GetOrderDiscounts** з екземпляра **cust** класу **Customer** з **лістингу 3.7**. Показаний і рядок коду, який

фактично здійснює цей виклик. Ви можете виконати подвійне клацання мишею, щоб перейти у вікно редактора коду якраз до цього твердження. На практиці, подвійне клацання мишею по будь-якій з точок виклику у вікні **Call Hierarchy** переміщає вас у вікно редактора якраз до того рядка коду, який містить цей виклик.

Вікно **Call Hierarchy** *показує всі можливі шляхи, якими можна вийти до конкретного рядка коду*. Хоча ця можливість дуже корисна, вона *обмежена статичним представленням* вашого коду і не надає можливості детального перегляду коду працюючої програми, яка може знадобитися для цілей налаштування. При налаштуванні вам зазвичай потрібно бачити робочий стан програми в конкретний момент її виконання. Далі ми розглянемо різні функції відлагоджувача, які допомагають досліджувати поведінку коду програми під час її виконання.

3.13.3. Конфігурація налагоджувального режиму

За замовчуванням, **VS** створює проекти з активізованим режимом налаштування, у якому вказані налаштування проекту, завдяки яким і стає можливим налаштування вашого застосування. Панель інструментів **VS** показує поточні конфігураційні налаштування, які ви використовуєте в даний момент. У списку, який розкривається наявна *налагоджувальна* (**Debug**) і *звичайна* (**Release**) конфігурації. Звичайна конфігурація (**Release**) визначає налаштування, які ваша програма матиме, коли ви почнете її розповсюдження для фактичного використання споживачами у виробничих умовах. Крім того, ви можете створити й індивідуально налагоджену конфігурацію, в якій можна задати саме ті властивості проекту, які вам потрібні. Тут ми будемо використовувати налагоджувальну конфігурацію (**Debug**).

Щоб вивчити можливості, які надаються налагоджувальною конфігурацією (**Debug**), переконайтеся в тому, що вона вибрана в інструментальній панелі. Щоб зробити це, потрібно, щоб був відкритий який-небудь проект. Потім виконайте подвійне клацання мишею по теці **Properties** вашого проекту у вікні **Solution Explorer** і перейдіть на вкладку **Build**, як показано на **рис. 3.45**.

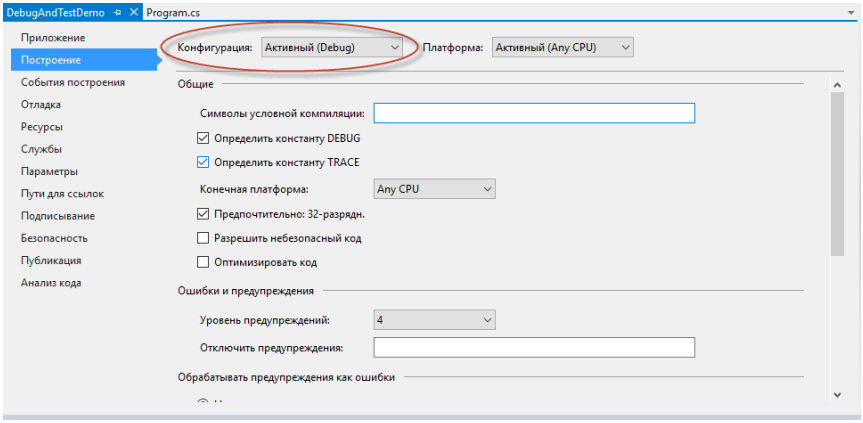


Рис. 3.45. Вкладка **Build (Построение)** вікна властивостей проекту

Як показано на **рис. 3.45**, *оптимізація для проекту відключена*, і визначені обидві змінні компілятора – **TRACE** і **DEBUG**. Якщо *включити оптимізацію*, то компілятор виконуватиме додаткову обробку коду, у тому числі, його структурні зміни. У результаті такої оптимізації скомпільований код буде компактніший за розмірами і виконуватися буде швидше. При *налаштуванні оптимізація не потрібна*, тому що ви виконуватимете код покроково і стежити за виведенням компілятора. Константи компілятора (також відомі під назвою *директив компілятора* – **compiler directives**), такі, як **TRACE** і **DEBUG**, використовуються компілятором для активізації або блокування окремих блоків коду. Наприклад, простір імен **System.Diagnostics** має клас **Debug**, який застосовуватиметься тільки у тому випадку, якщо визначена константа компілятора **DEBUG**.

Виконайте побудову вашого застосування, внаслідок чого буде отримано ряд файлів, придатних для використання в налагоджувальних цілях. Щоб переглянути ці файли, виконайте клацання правою кнопкою миші на імені рішення, проекту або теки у вікні **Solution Explorer** і виберіть з контекстного меню команду **Open Folder in Windows Explorer**. Потім перейдіть у вкладену теку **bin\Debug**, яка виглядатиме приблизно так, як показано на **рис. 3.46**.

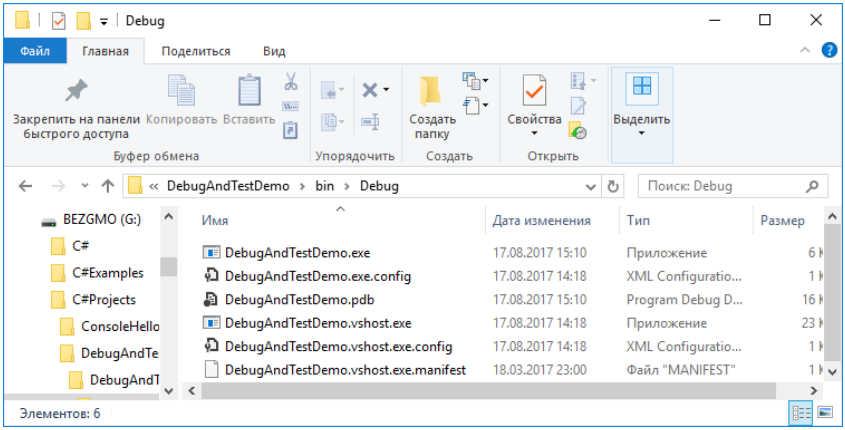


Рис. 3.46. Вміст теки **bin\Debug** після побудови проекту

На **рис. 3.46** ви побачите чотири файли, два з яких є файли застосування, а інші два призначені для підтримки роботи застосування під відлагоджувачем. Файл **DebugAndTestDemo.exe** є виконуваним файлом консольного застосування, поява якого цілком очікувана (у разі успішної побудови). Файл ***.pdb** є *файлом з налагоджувальною інформацією* (**symbol file**), який допомагає синхронізувати ідентифікатори у вашому коді з виконуваним файлом, спрощуючи тим самим покрокове виконання коду відлагоджувачем.

Файли, у складі імен яких присутній рядок **.vshost** призначені для забезпечення процесу налаштування. Файл ***.vshost** забезпечує швидше завантаження вашого застосування під час налаштування, дає вам можливість тестування вашого застосування з використанням різних конфігурацій безпеки, а також дозволяє оцінювати вирази в ході налаштування. Файли, імена яких містять рядок **.vshost**, призначені виключно для налагоджувальних цілей, тому *ви не повинні постачати їх з вашим застосуванням*. Кінцевому користувачеві вони ні до чого і тільки займатимуть дисковий простір. Зазвичай файли з рядком **.vshost** у складі імен потрібні тільки для цілей налаштування за допомогою **VS**. Існують різні параметри налаштування налагоджувального режиму, які ви можете конфігурувати у **VS**. Вони впливають на сеанс налаштування і модифікують налаштування конфігураційних файлів **.vshost**. Щоб задати налаштування режиму налагодження, відкрийте вікно

властивостей проекту і перейдіть на вкладку **Debug**, показану на **рис. 3.47**.

У вікні на **рис. 3.47**, опція **Configuration** встановлена на **Active (Debug) (Активный (Debug))**, а зі списку **Platform** вибрана платформа **Any CPU**. Зі списку **Platform**, який розкривається, можна вибрати наступні опції: **Any CPU, x86, x64** або **Itanium** – залежно від типу процесорної платформи, для якої виконується побудова застосування. Компілятор виконуватиме оптимізацію коду для вибраного вами типу **CPU**. Якщо ви запускаєте **VS** під управлінням **64**-розрядної **ОС** (це наш випадок), то в цьому полі може відображатися опція **Active (Any CPU)**.

Група опцій **Start Action** на вкладці **Debug** вказує, яким чином розпочинається сеанс налаштування. За замовчуванням пропонується опція **Start Project**. Опція **Start External Program** дозволяє *підключати налагоджувальний сеанс* **VS** до вже працюючого застосування, а опція **Start Browser With URL** *дає можливість налаштування* **Web**-застосувань. У загальному випадку, *для налаштування застосувань*, призначених *для роботи на настільних комп'ютерах*, вам буде потрібна тільки пропонована за замовчуванням опція **Start Project**. Для **Web**-застосувань, які автоматично запускаються у браузері, сторінка властивостей міняється.

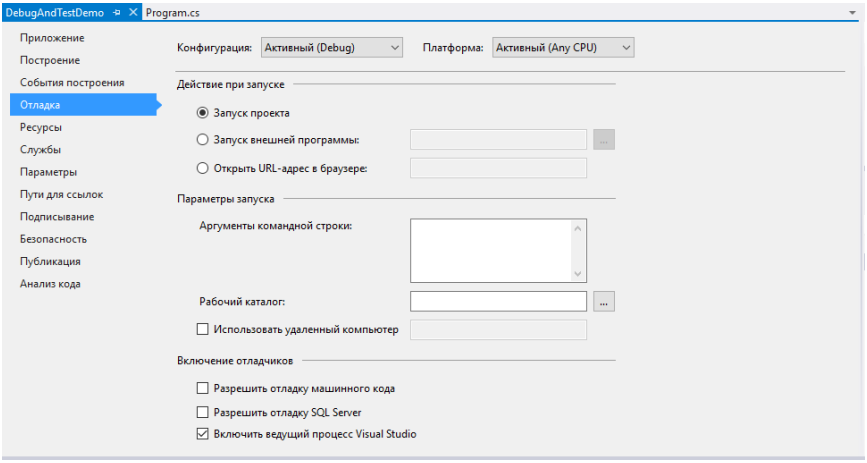


Рис. 3.47. Вкладка **Debug** вікна властивостей проекту

У групі **Start Options (Параметры запуска)** ви можете додати в поле **Command line arguments (Аргументы командной строки)** *список аргументів командного рядка, розділених пропусками*. Якщо ви пишете застосування, яке повинне запускатися в сеансі командного рядка або з командного сценарію (*скрипта*), то ця можливість буде дуже корисна для тестування і налаштування різних комбінацій аргументів командного рядка. У цьому випадку ви зможете прочитати значення аргументів командного рядка, які ви ввели в текстове поле **Command line arguments**, прочитавши їх з масиву аргументів, переданих методу **Main**.

Робоча тека – це коренева тека, з якої ваша програма читає файли і куди виводить їх, отримані в результаті її роботи. За замовчуванням, для *налагоджувальних конфігурацій* це буде тека **bin\Debug**, а для *робочих конфігурацій* – тека **bin\Release**. Ви можете змінити робочу теку, ввівши шлях до потрібної вам теки в полі **Working Directory (Рабочий каталог)**.

Прапорець **Use Remote Machine (Разрешить отладку ашинного кода)** призначений для складніших сценаріїв налаштування, при яких *можна налаштовувати застосування, яке працює на віддаленому комп'ютері*. Щоб користуватися цією можливістю, *треба буде встановити на віддаленому комп'ютері спеціальне програмне забезпечення для віддаленого налаштування*. Потім треба буде переконатися в тому, що в полі **Output path** вкладки **Build** вікна властивостей вказаний правильний шлях до виконуваного файла застосування, яке треба налаштовувати, що ця тека для виведення надана в спільний доступ, і що застосування має всі права, потрібні для доступу до цієї розділюваної теки.

Ми основну увагу приділимо *керованому коду* (**managed code**), який працює на **.NET CLR**. **VS** має можливість налаштування і *некерованого коду* (**unmanaged code**), наприклад, коду, написаного на C++ і який безпосередньо взаємодіє з **ОС**. У загальному випадку, треба залишати прапорець **Enable unmanaged code debugging**у скинутому стані, за виключенням тих ситуацій, коли ви пишете *керований код*, який взаємодіє з *некерованим кодом*, скажімо, бібліотекою **COM DLL**, і вам потрібна можливість налаштування як керованого коду, так і некерованого. Крім того, **VS** *надає можливість відкривати збережувані процедури* **SQL Server**,

встановлювати точки зупину і покроково виконувати код збережуваної процедури, в налагоджувальних цілях. *Якщо ви плануєте налаштування збережуваних процедур, встановіть цей прапорець*.

Прапорець **Enable the Visual Studio hosting process,** є саме тим налаштуванням, встановлення якого приводить до створення файлів **.vshost** у каталозі виведення. За звичайних умов цей прапорець бажано залишити у втановленому стані, через описані трохи раніше переваги, які надаються файлами **.vshost**. Єдиним виключенням може бути ситуація, в якій сервіси, які надаються файлами **.vshost**, конфліктують з кодом, який ви виконуєте. Але це – сценарій дуже складний і украй рідкісний.

На додаток до налаштувань налагоджувального режиму у вікні властивостей, велике число опцій налаштування буде вам доступне і у вікні **Options**. Щоб проглянути ці опції, виберіть з меню команди **Tools | Options (Свойства | Параметры)** і в лівій панелі вікна, яке розкрилося виберіть вузол **Debugging (Отладка)** (**рис. 3.48**).

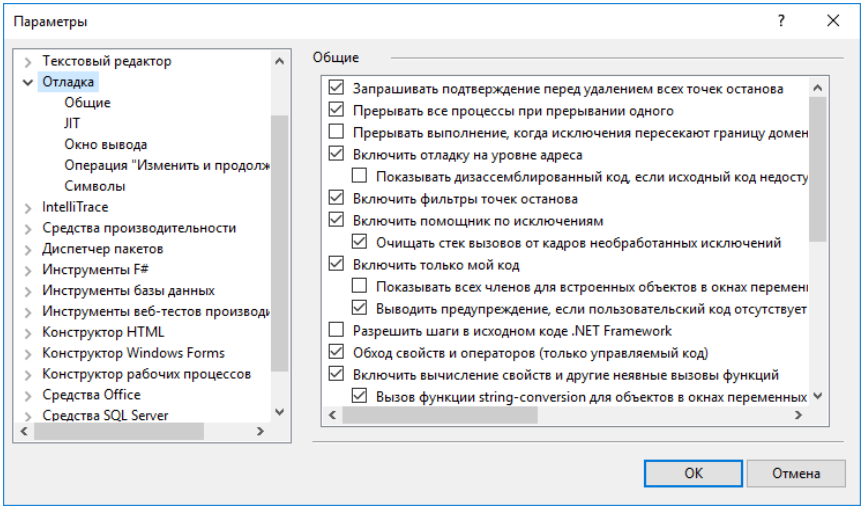


Рис. 3.48. Налагоджувальні опції у вікні **Options**

Як видно з **рис. 3.48**, існує велике число опцій, які дозволяють вам конфігурувати процес налаштування. Основна відмінність цих

опцій від опцій, специфічних для проекту, полягає в тому, що опції проекту діють і стосуються тільки цього проекту, тоді як опції налаштування, які задаються у вікні **Options**, дозволяють вам задати їх для усіх проектів. У тих випадках, коли це можливо, опції налаштування, які задаються у вікні **Options**, пропонуватимуться за замовчуванням і для всіх новостворюваних проектів. Отже, якщо ви хочете, щоб деякі опції за замовчуванням пропонувалися для усіх ваших проектів, в першу чергу встановіть їх у вікні **Options**. Опції налагоджувального режиму, які надаються у вікні **Options** (див. **рис. 3.48**), занадто численні, багато з них відносяться до складних сценаріїв і виходять за рамки питань, які ми будемо розглядати. Тому перерахувати їх всі не є доцільним. Якщо коли-небудь у вас виникне питання про те, чи доступна у VS та або інша можливість, бажано просто розкрити вікно **Options** і спробувати її знайти.

Тепер, коли ваша система сконфігурована для налаштування, можна приступати до встановлення точок зупину і власне налаштування застосувань.

3.13.4. Встановлення точок зупину

Точки зупину (**breakpoints**) є такими твердженнями в складі вашого коду, на яких програма повинна автоматично робити паузу, аналогічно тому, як зупиняється відтворення музики або фільму, коли ви натискаєте кнопку **PAUSE**. Коли програма досягає встановленої точки зупину, її виконання призупиняється, і ви отримуєте можливість виконання налагоджувальних завдань. До числа налагоджувальних завдань відносяться перегляд значень змінних у цьому «замороженому» стані програми, оцінка виразів або редагування коду з наступним продовженням виконання.

3.13.4.1. Створення точки зупину

Щоб створити точку зупину, треба відкрити один з проектів, а потім відкрити у вікні редактора коду один з файлів, який містить початковий код програми. Як вже говорилося раніше, у використовуваних прикладах, ми використовуватимемо код тестового застосування, приведений у **лістингу 3.7**. У вікні редактора коду VS на лівій межі вікна редактора коду є широка смуга, яка трохи підсвічується сірим кольором. Якщо здійснити клацання мишею на цій межі, VS встановить точку зупину на

відповідному твердженні. Можна також клацнути мишею на потрібному вам твердженні, що передасть йому фокус введення, а потім натиснути клавішу **<F9>**. У результаті цих дій теж буде створена точка зупину. При цьому на межі з'явиться велика червона точка, а саме твердження, на яке встановлена точка зупину, буде виділена червоним підсвічуванням, як показано на **рис. 3.49**.

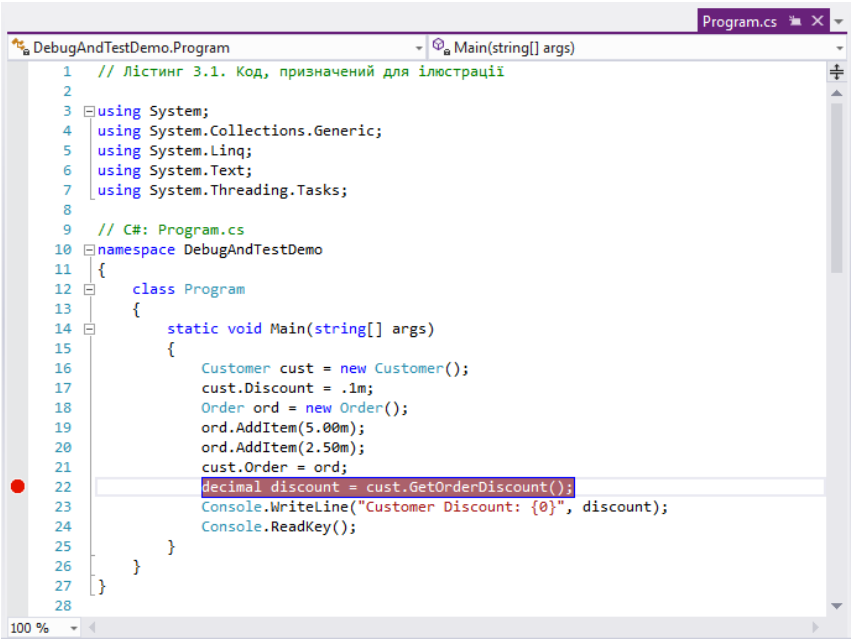


Рис. 3.49. Точка зупину

Зверніть увагу, що *точку зупину можна встановити тільки на код, який дійсно виконується* в ході роботи програми. Якщо ви виберете рядок, який не виконується (прикладом такого рядка є, наприклад, визначення простору імен), то точка зупину встановлена не буде, а в нижній частині екрану з'явиться повідомлення про те, що у цій позиції точку зупину створити неможливо («**A breakpoint could not be inserted at this location**»).

Щоб переконатися в тому, що VS зробить паузу, зустрівши точку зупину, запустіть застосування в режимі налаштування. Щоб почати виконання програми в режимі налаштування, можна вибрати з меню команди **Debug | Start Debugging**. Як варіант, можна натиснути клавішу **<F5>** або ж клацнути кнопку інструментальної панелі **Start With Debugging** із зображенням невеликої трикутної зеленої стрілки, спрямованої вправо. Точка зупину на **рис. 3.49** встановлена на виклику методу **GetOrderDiscount** в методі **Main**. Коли програма зустрічає точку зупину, відповідний рядок коду підсвічується жовтим кольором, а в центрі червоної крапки, що позначає точку зупину, з'являється жовта стрілка. Клацання мишею по кнопці **Continue** (це та ж сама кнопка, яка використовувалася для запуску програми в режимі налаштування) або натиснення клавіші **<F5>** приведуть до того, що VS продовжить виконання коду програми. Зупинити процес налаштування теж можна декількома різними способами. *По-перше*, можна вибрати з меню команди **Debug | Stop Debugging**, *по-друге*, можна натиснути клавіатурну комбінацію **<SHIFT+F5>** і, нарешті, можна клацнути мишею кнопку інструментальної панелі **Stop Debugging** із зображенням невеликого кольорового квадрата.

3.13.4.2 Індивідуальне налаштування точки зупину

У попередньому розділі були викладені базові відомості про точки зупину. Крім того, було показано, як встановити точку зупину на потрібному рядку коду. Проте це далеко не все, що треба знати про точки зупину. Наприклад, їх можна індивідуально налаштувати так, щоб програма зупинялась на ній залежно від певних критеріїв – наприклад, від числа раз, коли ця точка зупину була зустрінута (**hit count**), від виконання або невиконання конкретної умови і т. д. Щоб проглянути, які опції вам доступні, наведіть курсор на одну з точок зупину, клацніть правою кнопкою миші, і ви побачите повний набір доступних опцій в контекстному меню, яке з'явилося. Опції цього контекстного меню коротко описані в **таблиці 3.1**.

Таблиця 3.1. Опції, доступні в контекстному меню точки зупину

Опція	Опис
Delete	Видаляє точку зупину.

Breakpoint	
Disable/Enable Breakpoint	Якщо ви не хочете видаляти точку зупину, тому що згодом плануєте використати її повторно, ви можете блокувати її, а потім знову активізувати, коли вона знову знадобиться.
Location	Це – той вид точок зупину, коли виконується клацання на межі вікна редактора коду. Ви можете змінити параметри цієї точки зупину, вибравши з контекстного меню команду **Location** і відредагувавши параметри в діалоговому вікні, яке з'явилося.
Condition	Дозволяє ввести вираз, залежно від значення якого програма зупинятиметься, зустрівши цю точку зупину. Програма зупинятиметься, або якщо вираз отримує значение **TRUE**, або якщо змінюється значення змінної. Вираз повинен базуватися на змінних, які зустрічаються у вашому коді.
Hit Count	Приводить до того, що програма почне зупинятися на цьому рядку після того, як він буде виконаний задане число разів, або коли лічильник кратний деякому числу (т. т. кожен **n**–й раз), або коли лічильник досягне або перевищить деяке значення.
Filter	Точка зупину спрацьовуватиме (т. т. виконання програми буде зупинятися) тільки тоді, коли зустрінеться вказана вами комбінація машини (**machine**), процесу (**process**) або нитки/потоку виконання (**thread**).
When Hit	Задає точку трасування (**tracepoint**), яка виводить повідомлення у вікно виводу. Повідомлення можна конфігурувати таким чином, щоб до його складу були включені різні системні значення, наприклад, функції (**function**), потоки управління (**thread**) і т. д. Ви можете переглядати повідомлення у вікні **Output**. Щоб скористатися цією можливістю,

	виберіть з меню команди **View | Output Window** або натисніть клавіатурну комбінацію **<CTRL>+<ALT>+<O>**. У момент спрацьовування точки зупину ви можете запускати макрос.
Edit Labels	Ви можете асоціювати точки зупину з мітками, щоб розподілити їх по групах.
Export	Ця опція дозволяє експортувати точки зупину у зовнішній файл формату **XML.**

Ви можете встановити функціональну точку зупину, клацнувши по методу, щоб його виділити, а потім вибравши з меню команди **Debug | New Breakpoint | Break At Function (Отладка | Создать точку останова | Прервать в функции…)** або натиснувши клавіатурну комбінацію **<CTRL>+<D>, <N>**.

3.13.4.3 Управління точками зупину

У міру роботи над кодом вашої програми, в ньому з'являється все більше і більше точок зупину, розкиданих по усьому вашому проекту. Можна управляти усіма точками зупину централізовано з єдиного вікна. Для цього виберіть з меню команди **Debug | Windows | Breakpoints (Отладка | Окна | Точки останова)**, внаслідок чого з'явиться вікно, показане на **рис. 3.50**.

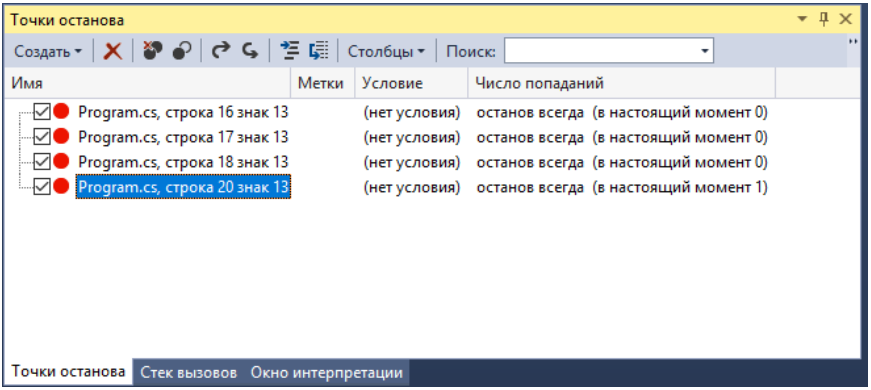


Рис. 3.50. Вікно **Breakpoints**

Більша частина функцій, доступних у вікні **Breakpoints**, вже пояснена, за винятком того факту, що опції інструментальної панелі діють стосовно до усіх точок зупину, які встановлені і активізовані на поточний момент. Клацання мишею по заголовку стовпця призводить до сортування його вмісту. Поле пошуку **Search (Поиск)** допомагає фільтрувати точки зупину, а список **Columns (Столбцы)** дозволяє розставити акценти на тих полях, по яких повинен здійснюватися пошук. Крім того, на інструментальній панелі вікна **Breakpoints** є кнопки **Export()** та **Import()**, які дозволяють експортувати точки зупину у зовнішній **XML**-файл і відповідно, імпортувати їх із зовнішнього **XML**-файла. Подвійне клацання мишею по будь-якій точці зупину з числа наявних у вікні **Breakpoints** переміщає вас до рядка коду у вікні редактора коду, на який встановлена ця точка зупину. Клацання правою кнопкою миші по точці зупину відображає контекстне меню, яке містить опції, які вже нами обговорювалися раніше.

Після встановлення точок зупину, ви можете покроково виконувати код програми, вивчаючи послідовність її виконання.

3.13.4.4 Покрокове виконання коду

Покрокове виконання коду є процесом виконання одного або декількох рядків коду, яке відбувається під контролем програміста, який виконує налаштування. На деталізованому структурованому рівні, ви можете виконувати цей процес не просто покроково, а рядок за рядком. Хоча такий режим часто буває потрібний для з'ясування того, де ж «зачаїлася» помилка, не менш часто він буває занадто трудомісткий і незграбний. Тому і були передбачені методи «перескакування» через декілька рядків коду, за рахунок їх виконання за один прийом.

Щоб почати покрокове виконання коду, відкрийте проект, встановіть точку зупину і запустіть програму на виконання в режимі налаштування. Програма запуститься, і її виконання продовжиться доти, поки не зустрінеться точка зупину. У цей момент часу виконання програми буде перерване, і ви зможете виконати різні налагоджувальні операції: **Step, Step over** і **Step out**. Доступні налагоджувальні операції коротко перераховані в **таблиці 3.2**. Пояснення, запропоновані в цій таблиці, зроблені в припущенні того,

що в ході виконання програми зустрілася точка зупину, і в цей момент її виконання перерване, і подальший процес залежатиме від того, яку з операцій ви виберете.

Таблиця 3.2. Покрокові операції

Операція	Опис
Step Over	Виконує код поточного рядка і переміщається до наступного рядка коду, після чого виконання програми знову зупиняється, і відлагоджувач чекає ваших подальших інструкцій. Щоб виконати цю операцію, виберіть з меню команди **Debug | Step Over**. Альтернативні варіанти виконання цієї операції – натиснення клавіші **<F10>** або кнопки **Step Over()** на інструментальній панелі. Окрім того, можна виконати клацання правою кнопкою миші і вибрати відповідну опцію з контекстного меню. Більшість розробників Visual Studio швидко запам'ятовують «гарячу клавішу» **<F10>** і віддають перевагу саме їй.
Step Into Specific	Якщо *поточний рядок виконує виклик методу*, то команда **Step Into** передасть управління першому рядку методу, який викликається, і виконання програми призупиниться там. Щоб виконати цю операцію, виберіть з меню команди **Debug | Step Into**. Альтернативні варіанти полягають в натисненні клавіші **<F11>** або в клацанні мишею по кнопці **Step Into** на інструментальній панелі. Практика показує, що клавіша **<F11>** надає найшвидший варіант виконання цієї операції.
Step Out	Якщо ви знаходитеся всередині методу, ви можете повернутися назад до викликаючого коду, виконавши операцію **Step Out**. Щоб виконати операцію **Step Out**, або виберіть з меню команди **Debug | Step Out**, або натисніть клавіатурну комбінацію **<SHIFT>+<F11>**, або клацніть мишею по кнопці **Step Out** на інструментальній панелі. Зверніть увагу, що всередині функції не пропускається жодний рядок коду; вони

	виконуються в точній відповідності з логікою вашої програми. Виконання програми автоматично призупиниться на рядку коду, який йде за поверненням з цієї функції.
Run to Cursor	Іноді вам може знадобитись виконати блок коду і зупинитися на певному рядку. На цьому рядку можна встановити іншу точку зупину і продовжувати виконання програми доти, поки не буде досягнута нова точка зупину. Проте якщо ви не бажаєте зберігати нову точку зупину на тому рядку, де бажаєте зробити паузу у виконанні програми, можна клацнути правою кнопкою миші по цьому рядку і вибрати з контекстного меню, що з'явилося, команду **Run To Cursor**. Знову ж таки, тут не пропускається жоден з рядків коду, програма тільки зробить паузу, коли її виконання досягне рядка, на який ви помістили курсор. Як варіант, ви можете виділити рядок, на якому хочете зробити паузу, і натиснути клавіатурну комбінацію **<CTRL>+<F10>**. Це особливо корисно, якщо у вас виникла підозра, що виконуються усі ітерації циклу.
Set Next Statement	Ви можете пропускати декілька рядків коду, «перестрибуючи» через них як вперед, так і назад, без виконання пропущеного коду. Наприклад, може виявитися корисним пропустити метод, щоб з'ясувати, що насправді ви хотіли увійти до нього. Це дозволяє повернутися до рядків коду, починаючи з яких вам хочеться простежувати логіку роботи застосування, адже якщо перезапуск застосування насправді не потрібний, то і не захочеться його виконувати. Щоб повернутися до рядка коду, звідки ви хочете перейти до виклику методу, виділіть жовту стрілку на межі і перемістите її назад до виклику методу, після чого ви зможете виконати команду **Step Into**. Як варіант, якщо у вас є одне або декілька тверджень, які ви не хочете виконувати, перетягніть жовту стрілку на межі до твердження, яке йде відразу ж за кодом, що

	пропускається, і продовжуйте покроковий налагоджувальний сеанс. Цей підхід особливо зручний, якщо ви використовуєте функцію **Edit and Continue**, яка дозволяє редагувати код програми «на льоту», експериментувати з різними ідеями з програмування і миттєво повторно виконувати ці рядки коду. Зверніть увагу, що VS не скидає значення змінних до початкового стану, тому, щоб отримати очікувані результати, вам треба скинути ці значення вручну.

Операція **Step Over** виконує код поточного рядка і переходить до наступного. Операцію **Step Over** можна виконати вибравши з меню команди **Debug | Step Over**, натиснувши на клавіатурі клавішу **<F10>** або клацнувши мишею по кнопці **Step Over()** на інструментальній панелі.

Отже, ви навчилися виконувати покрокове налаштування коду, що *надзвичайно корисно*. Проте не менш важливо мати можливість перегляду значень змінних і спостерігати за їх зміною.

3.13.5. Дослідження стану застосування

Стан або *статус застосування* (**application state**) – це *значення всіх змінних* у вашому коді, *поточний шлях виконання*, а також *будь-яка інформація, яка повідомляє* вам про те, що робить ваша програма. Можливість перегляду інформації про стан програми під час налаштування дуже важлива, тому що вона дозволяє виявити відмінності між тим, що програма робить насправді, і тим, що ви від неї чекали. У VS є багато різних вікон (ми про це говорили раніше), які дозволяють переглядати інформацію про стан застосування, в цьому підрозділі ми розглянемо їх ще раз стосовно коду прикладу.

> **Примітка:** При дослідженні статусу застосування, треба буде враховувати концепцію *діапазону* (**scope**). Коли змінна знаходиться в межах діапазону, то можна переглянути її значення. Діапазон визначається в межах блоку. У C# блоки визначаються фігурними дужками. Прикладами діапазону є *поля класу* і *локальні змінні*. *Поле приватного класу* знаходитиметься в межах діапазону для всіх методів цього

класу, але не у іншому класі. *Локальна змінна* знаходитиметься в межах діапазону для усіх методів, в яких вона визначена, але знаходитиметься поза діапазоном для інших методів. Ще один приклад діапазону – змінна циклу, яка буде знаходитися в межах діапазону в тілі циклу, але за його межами – поза полем циклу.

3.13.5.1. Вікна Locals і Autos

Вікна **Locals (Локальные)** та **Autos (Видимые)** показують значення змінних для поточної точки зупину. Вікно **Locals** містить список усіх змінних в межах діапазону, до яких може отримувати доступ поточне твердження. Вікно **Autos** показує змінні з поточного і попередньою рядків коду. Можна відкрити вікна **Locals** та **Autos**, при активному сеансі налаштування VS тоді, коли виконання програми призупиняється досягши точки зупину. Ці вікна можуть вже бути присутніми на екрані в нижньому лівому кутку, розмістившись за вікном **Output**, якщо середовище розробки VS не переналаштовувалось.

Як показано на **рис. 3.51**, вікно **Locals** показує всі змінні в діапазоні видимості методу **Main** з **лістингу 3.7**.

Вікно **Locals** відображає змінні у великомодульному представленні, і список може бути досить довгим, залежно від того, яке число змінних знаходиться в діапазоні видимості. Вікно **Locals** *зручно використовувати* для того, щоб з'ясувати, які змінні перебувають під впливом поточного алгоритму. **На рис. 3.52** показане вікно **Autos**.

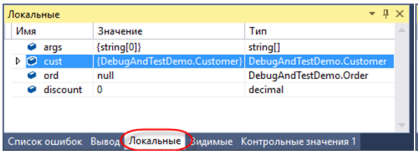


Рис. 3.51. Вікно **Locals**

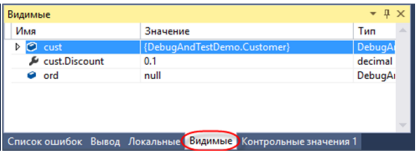


Рис. 3.52. Вікно **Autos**

Зверніть увагу на те, що вікно **Autos** відображає дані у більш дрібномодульному представленні, і це стосується як змінних, так і властивостей об'єктів з *поточного* і *попереднього* рядків коду. Вікно **Autos** зручно використовувати для більш цілеспрямованого з'ясування подій, які на даний момент відбуваються у вашому коді.

3.13.5.2. Вікно Watch

Вікно **Watch (Контрольные значения)** дозволяє створити індивідуальний список змінних, за якими вам потрібно вести спостереження. Ви можете перетягувати змінні мишою з вікна редактора коду або вручну вводити імена змінних у вікно **Watch**. Виберіть з меню команди **Debug | Windows | Watch**, і ви побачите список з чотирьох вікон **Watch**, у яких ви можете створити чотири різних набори даних для перегляду по одному набору за раз. На **рис. 3.53** показано вікно **Watch**.

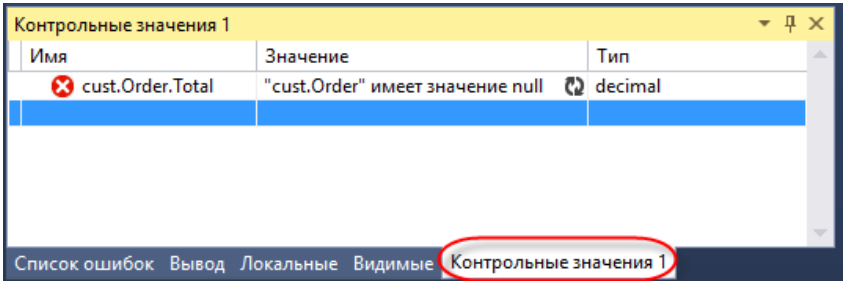


Рис. 3.53. Вікно **Watch**

Вікна **Locals** та **Autos** можуть виявитися захаращеними занадто великим числом змінних і уповільнюватимуть вашу роботу по мірі ускладнення вашого коду, особливо коли змінні, які цікавлять вас знаходяться в самому кінці списка або так далеко один від одного, що для їх перегляду вам доведеться прокручувати список. Ось тут вам і прийде на допомогу вікно **Watch** – з його допомогою ви зможете швидко дістатися до об'єкта без потреби постійного розгортання ієрархічного дерева. Наприклад, ви можете ввести у вікно рядок **cust.Order.Total**, як показано на **рис. 3.53**, щоб переглянути результуюче значення властивості **Totals** властивості **Order** екземпляра **cust**.

Крім того, в цьому вікні можна редагувати значення ваших змінних і властивостей. Для цього в ньому треба або виконати подвійне клацання мишею по поточному значенню в стовпці **Value,** або виконати клацання правою кнопкою миші по імені змінної і вибрати з контекстного меню команду **Edit**. Коли значення буде змінено, колір, яким воно відображається, зміниться з чорного на

червоний, щоб звернути вашу увагу на зміну. Цей спосіб редагування значень «на льоту» дуже зручний, особливо якщо ви хочете повторно виконати попередні рядки коду з новим значенням, не перериваючи сеансу налаштування і не перезапускаючи програму. Ці методи роботи дозволяють досягти суттєвого заощадження часу.

3.13.5.3. Вікно Immediate

При налаштуванні часто буває корисно оцінювати значення виразів на поточний момент. У вікні **Immediate (Окно интерпретации)** можна вводити імена змінних і вирази різних типів. Щоб розкрити вікно **Immediate**, виберіть його з меню команди **Debug | Windows**. Крім того, залежно від опцій налаштування VS, це вікно вже може бути відкрите в правому нижньому кутку екрана налаштування. Вікно **Immediate** показане на **рис. 3.54**.

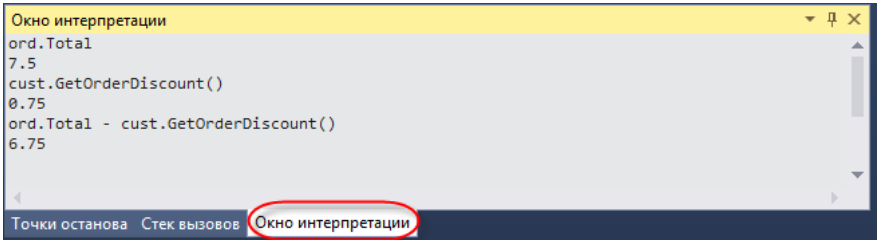


Рис. 3.54. Вікно **Immediate**

У прикладі, представленому на **рис. 3.54**, у вікні **Immediate** показані три вирази, які демонструють доступні вам можливості з читання властивостей, і заповнення методів та оцінці виразів. У цьому прикладі ці вирази були введені вручну, і ви можете робити те ж саме, вводячи практично будь-який код, який вам потрібно.

3.13.5.4. Вікно Call Stack

Як вже говорилося раніше, при обговоренні інструментарію, який застосовується під час розробки, у вікні **Call Hierarchy** вам надається можливість перегляду коду, який викликає ваші методи викликів під час розробки. Отже, ви маєте аналогічну *можливість перегляду шляху виконання коду під час виконання програми* – для цього служить вікно **Call Stack**. Під час сеансу налаштування ви

знайдете вікно **Call Stack** в правому нижньому кутку VS на вкладці, розташованій поруч з вікном **Immediate**, якщо тільки ви не змінили первинне налаштування середовища VS. Інакше, ви можете відкрити це вікно, вибравши з меню команди **Debug | Windows | Call Stack**. У вікні **Call Stack** ви можете переглянути поточний шлях виконання вашого застосування від методу **Main** до поточного рядка коду. На **рис. 3.55** показано вікно **Call Stack**. Щоб зрозуміти, чому воно називається **Call Stack**, зверніть увагу на те, що кожен наступний метод як би «поміщається в стек» поверх попереднього. При цьому поточний метод знаходиться на вершині стека, а *точка входження* – в самому низу, а наступні виклики – між ними. Цей «стек викликів» можна порівняти, наприклад, зі стопкою тарілок, де остання тарілка знаходиться вгорі.

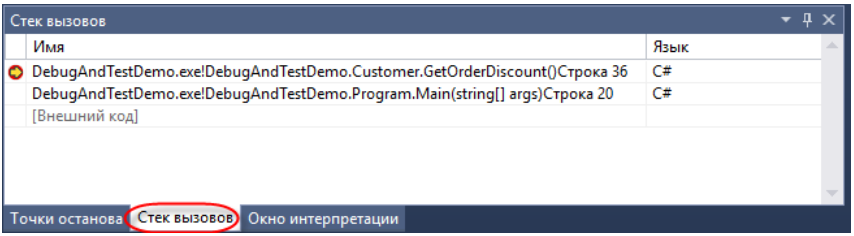


Рис. 3.55. Вікно **Call Stack**

У вікні **Call Stack**, показаному на **рис. 3.55**, видно, що в процесі налаштування здійснено входження в метод **GetOrderDiscount**. Подвійне клацання мишею по іншому методу у вікні **Call Stack** переміщає вас до точки виклику, з якої викликався цей метод. Це – дуже важливий і дуже потужний інструмент, тому що він дозволяє вам перейти до викликаючого коду і досліджувати стан застосування у точці виклику, надаючи цінну візуальну інформацію про те, які обчислення здійснювалися перед викликом поточного методу і як вони були сформульовані.

3.13.5.5. Вікно QuickWatch

Вікно **Quick Watch (Быстрая проверка)** дає можливість швидкого перегляду виразу. При введенні виразу воно пропонує підтримку **Intellisense**, дозволяючи повторно виконати оцінку виразу і додати вираз у вікно **Watch**. Щоб відкрити вікно **Quick Watch**,

виберіть з меню команди **Debug | Quick Watch (Отладка | Быстрая проверка)** або натисніть клавіатурну комбінацію **<CTRL>+<D>, <Q>**. Якщо ви виберете вираз у редакторі коду, то вікно **QuickWatch** відобразить цей вираз. На **рис. 3.56** показано типове вікно **QuickWatch** і його використання для оцінки виразу.

Якщо клацнути мишею по кнопці **Reevaluate (Пересчитать)**, показаній на **рис. 3.56**, то результати обчислення з'являться в стовпчику **Value**. Стовпчик **Value** міститиме тільки поточний вираз. Якщо ви хочете зберегти вираз, натисніть кнопку **Add Watch (Добавить контрольное значение)**, внаслідок чого вираз буде завантажено у вікно **Watch**. Треба мати на увазі, що якщо ви закриєте вікно **Watch**, вираз буде видалено, проте він буде присутнім у списку історії, з якого ви згодом зможете знову його вибрати.

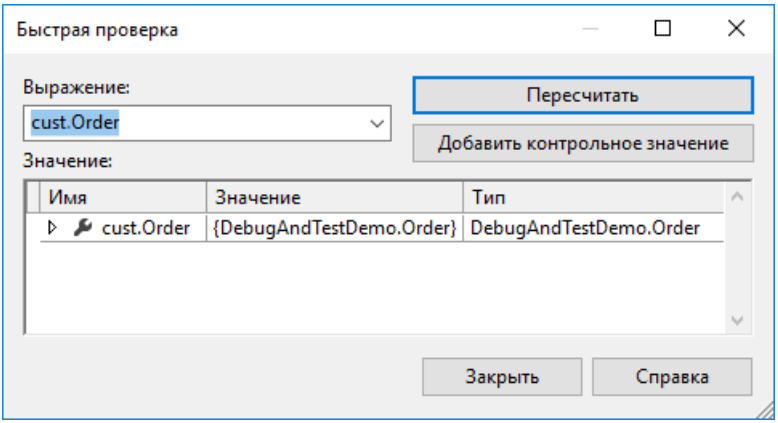


Рис. 3.56. Вікно **QuickWatch**

3.13.5.6. Спостереження змінних з прив'язкою до джерела

У процесі налаштування, ви можете затримати курсор над будь-якою змінною, щоб побачити її значення. Але, як тільки ви відведете курсор убік, зникне і спливаюче вікно підказки, яке відображає значення змінної. Функція **Pin To Source (Прикрепить к источнику)** йде на крок далі, дозволяючи виводити значення змінної на постійній основі. Щоб використати функцію **Pin To Source**, клацніть правою кнопкою миші по змінній і виберіть опцію **Pin To Source**. Як альтернативний варіант, ви можете затримати курсор над

змінною у відлагоджувачі і клацнути мишею по значку із зображенням кнопки, який з'явиться разом зі значенням змінної у вікні спливаючої підказки. Результат прив'язки до джерела за допомогою функції **Pin To Source** показаний на **рис. 3.57**.

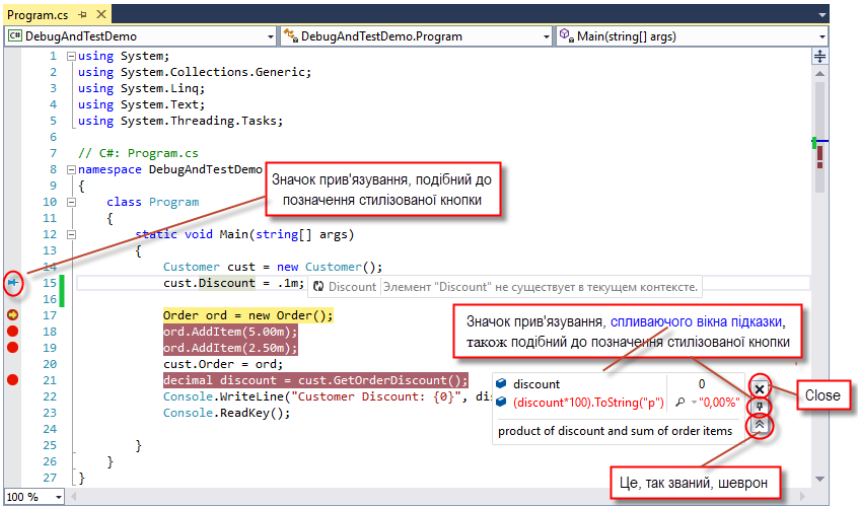


Рис. 3.57. Результат прив'язки значення змінної до джерела

Після того, як ви виконаєте прив'язку значення змінної до джерела, можна продовжувати налаштування і виконати прокручування назад до змінної, щоб прочитати її поточне значення. *На додаток до перегляду значення, ви можете додати коментар.* Для цього клацніть мишею по значку «шеврона», який з'явиться, коли ви затримаєте курсор над «прив'язаним» значенням. У цьому прикладі "прив'язане" значення має коментар «product of discount and sum of order items».

VS виявить «прив'язане» значення після рядка, але ви можете не побачити значення, якщо воно зустрічається в довгому рядку, який перевищує ширину вашого екрану. На щастя, ви можете клацнути мишею по «прив'язаному» значенню і перетягнути його в те місце екрану, де ви завжди зможете його бачити. Щоб запобігти плутанини, не забувайте все-таки розміщувати такі «прив'язані» значення найближче до тієї змінної, для якої вони відображаються.

Клацніть правою кнопкою миші по «прив'язаному» значенню, щоб відобразити контекстночутливе меню з опціями **Add/Remove Expression (Добавить/Удалить выражение)**. На **рис. 3.57** показано доданий вираз **(discount*100).ToString("p")**, щоб відобразити значення, виражене через відсотки. Додавання виразів дозволяє зробити вираз зручнішим для читання і сприйняття. Крім того, можна додавати інші вирази, пов'язані з даним, щоб мати можливість миттєвого перегляду і зрозуміти, як це значення впливає на інші розрахункові результати.

Закрити «прив'язане» значення можна, затримавши над ним курсор і клацнувши мишею по кнопці **Close (X)**.

3.13.5.7. Робота з IntelliTrace

Вікно **IntelliTrace** відображає список усіх змін, які відбулися *в ході сеансу налаштування*. У міру покрокового виконання коду, у вікні **IntelliTrace** відображається кожен крок сеансу налаштування. За допомогою інструментальної панелі вікна **IntelliTrace** ви можете встановити режим представлення інформації – **Diagnostic Events** (*діагностичні події*) або **Call View** (*перегляд викликів*). Режим перегляду діагностичних подій дозволяє фільтрувати події за *категоріями* (**Category**) і *нитками/потоками управління* (**Thread**). Клацання мишею по будь-якому з елементів, які відображаються у вікні **IntelliTrace**, дозволяє переглядати статус застосування на той момент часу, коли була зафіксована конкретна подія. Вікно **IntelliTrace** показане на **рис. 3.58**.

Вікно **IntelliTrace** *може бути дуже корисне*, якщо ви перейшли через твердження, яке змінило значення змінної, а потім вам знадобилося повернутися назад і подивитися, яким було значення змінної до того, як воно було змінене. На **рис. 3.58** показана якраз така ситуація, коли подія, яка підсвічується, **Breakpoint hit: Main**, дозволяє передивитися вікна **Locals** і **Call Stack**. Важлива відмінність у цьому випадку полягає в тому, що показані значення відносяться до моменту часу, коли сталася конкретна подія, а не до теперішнього моменту часу, що дає дуже важливу і цінну інформацію про те, що відбувалося в той момент з вашою програмою. Ще одна сфера застосування функції **IntelliTrace** полягає в тому, що з її допомогою можна вивчати файли журналів, створені іншим розробником або тестувальником ПЗ з допомогою нового

інструменту **Microsoft Test and Lab**, який може записувати увесь сеанс тестування.

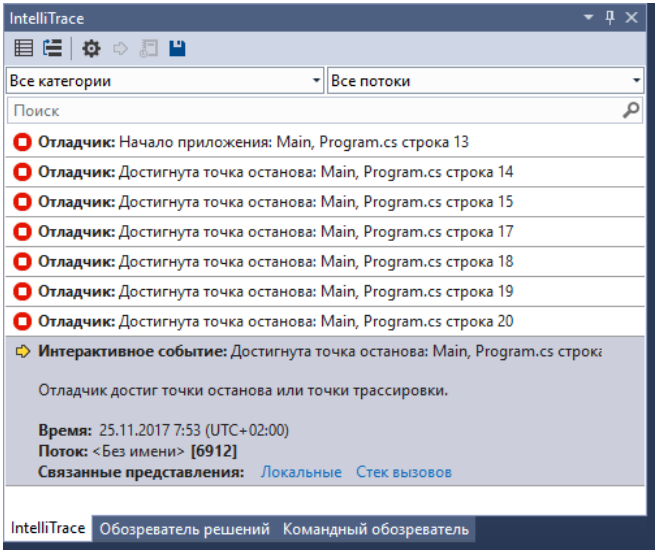


Рис. 3.58. Вікно **IntelliTrace** відображає історію сеансу налаштування

Щоб сконфігурувати опції **IntelliTrace**, треба вибрати з меню опції **Tools | Options | IntelliTrace (Отладка | Параметры | IntelliTrace)**. **IntelliTrace** створює файли журналів, які існують доти, поки запущене і працює середовище VS. При закритті VS, файли журналів видаляються, тому не забудьте скопіювати їх перед тим, як закрити VS. Розміщення файлів журналу задається в гілці **Tools | Options | IntelliTrace | Advanced (Отладка | Параметры | IntelliTrace | Дополнительно)**.

Якщо ви отримаєте файл журналу від іншого розробника, його можна завантажити, вибравши з меню команди **File | Open | Open New**. Після цього ви зможете передивитися історію налаштування і статус застосування для кожної події, яка мало місце в ході сеансу налаштування.

3.13.6. Рішення проблем за допомогою відлагоджувача VS

Зараз ми спробуємо розв'язати декілька практичних проблем, які демонструють, як треба застосовувати відлагоджувач VS для їх вирішення. Ми розглянемо дві найбільш типові ситуації, з якими може зіткнутися програміст у процесі повсякденної роботи: *виявлення та обробку некоректних даних і розв'язування проблеми нульових покажчиків* (**null references**). Сама по собі ця програма не особливо складна, але її логіка досить розвинена, щоб ви могли уявити, як можна заблукати в лабіринті і які підходи треба застосовувати, щоб звідти вибратися. Спочатку давайте розглянемо програму як таку, а потім виконаємо дві вправи на пошук та усунення помилок.

3.13.6.1. Програма з помилками

Код програми, яка розглядається у цьому розділі, містить помилки, і для виконання вправ важливо, щоб ви ввели цей код саме так, як він приведений в лістингах, нічого не змінюючи і не виправляючи, навіть якщо ви вже бачите ці помилки. Інакше вам не вдасться виконати описувані вправи. Ми детально опишемо кожен фрагмент коду, без спроби виявлення помилок-«секретів». Потім ми покроково пройдемо всю процедуру налаштування, з допомогою якої в реальному житті програмісти виконують пошук і виправлення помилок у своєму коді. Ця програма є засобом пошуку, який за введеним прізвищем шукає людину у списку клієнтів. Якщо програма знаходить ім'я в списку клієнтів, вона виводить прізвище та ім'я цього клієнта. Інакше, програма виведе повідомлення, яке інформує користувача про те, що вказане прізвище в списку клієнтів не виявлене.

Програма складається з трьох основних частин: класу, який містить інформацію про клієнта, класу, який повертає список клієнтів, і класу, який містить метод **Main**, який запускає програму. У наступних декількох розділах кожен з класів буде описаний детально.

Клас Customer

Кожного разу, коли вам потрібно працювати з даними, треба створити клас, який міститиме ці дані. Через те, що застосування

призначене для роботи з клієнтами, цілком природно буде назвати цей клас **Customer**. Код цього класу приведений в **лістингу 3.8**.

Лістинг 3.8. Код класу **Customer**

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

У цьому прикладі приведений лише потрібний мінімум інформації, призначений для демонстраційних цілей. Будь-яке застосування, яке має практичну цінність, повинне мати клас, в якому міститься набагато більше число властивостей. Зверніть увагу, що обидві властивості мають тип **string**.

Клас CustomerRepository

У нашій програмі ми створимо клас, єдине призначення якого полягає в роботі з даними. Це – типовий і широко поширений шаблон, який називається *репозитарієм* (**Repository**). У **лістингу 3.9** приведений код класу **CustomerRepository**, який має метод, що повертає список об'єктів **Customer**.

Лістинг 3.9. Код класу **CustomerRepository**

```
using System.Collections.Generic;
public class CustomerRepository
{
    public List <Customer> GetCustomers()
    {
        var customers = new List <Customer>
        {
                new Customer
                {
                        FirstName = "Семен",
                        LastName = "Кравчук"
                },
                new Customer
                {
```

```
                    FirstName = "Іван "
                },
                new Customer
                {
                    FirstName = "Віктор",
                    LastName = "Майструк"
                }
            };
            return customers;
        }
}
```

Метод **GetCustomers** повертає значення типу **List <Customer>**. Для подальшого продовження обговорення, те, як працює метод **GetCustomers**, великого значення не має. Такий метод зазвичай отримує список клієнтів з бази даних, від **Web**-сервісу або іншого об'єкта. Для простоти, **GetCustomers** ініціалізує список **List** об'єктами **Customer**. Особливо важливе значення має та частина методу, в якій властивість **FirstName** набуває значення **"Іван "**. Зверніть увагу на символ пропуску, доданий в кінець імені. Цей пропуск потрібний, щоб отримати саме такий сценарій поведінки програми, який задуманий (інакше кажучи, тут умисне зроблена помилка, яку ми усуватимемо в ході сеансу налаштування). Те, що для об'єкта **Customer** з властивістю **FirstName**, встановленою на **"Іван "**, відсутня властивість **LastName** – це теж «так задумано», щоб потім продемонструвати усунення цієї помилки.

Програма пошуку

У **лістингу 3.10** приведений код пошукової програми, яка використовує клас **CustomerRepository**, щоб отримати список об'єктів **Customer**. Програма крутиться в циклі, обробляючи список результатів, перевіряючи кожен член списка на відповідність заданому критерію пошуку. Якщо знайдений збіг, то програма виводить повне ім'я клієнта. Якщо збігів не знайдено, програма повідомляє про те, що пошук закінчився невдало (клієнт не знайдений).

Лістинг 3.10. Пошукова програма, яка повертає повне ім'я клієнта

```
using System;
class Program
{
	static void Main()
	{
		var custRep = new CustomerRepository();
		var customers = custRep.GetCustomers();
		var searchName = "Іван";
		bool customerFound = false;
		foreach (var cust in customers)
		{
			// 1. Перша помилка
			if (searchName == cust.FirstName)
			{
				Console.WriteLine(
					"Found: {0} {1}",
					cust.FirstName,
					cust.LastName);
					customerFound = true;
			}
		}
		if (!customerFound)
		{
			Console.WriteLine("Клієнта не знайдено.");
		}
		Console.ReadKey();
	}
}
```

Лістинг 3.11. Остаточний текст програми

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

namespace TestCustomersDemo
```

```
{
public class Customer
   {
public string FirstName { get; set; }
public string LastName { get; set; }
   }

public class CustomerRepository
   {
public List <Customer> GetCustomers()
      {
var customers = new List <Customer>
                  {
                        new Customer
                        {
                              FirstName = "Семен",
                              LastName = "Кравчук"
                        },
                        new Customer
                        {
                              FirstName = "Іван "
                        },
                        new Customer
                        {
                              FirstName = "Віктор",
                              LastName = "Майструк"
                        }
                  };
return customers;
      }
   }

class Program
   {
static void Main(string[] args)
      {
// Виведення без спотворень деяких букв українського алфавіту
Console.OutputEncoding = Encoding.GetEncoding(1251);
```

```
var custRep = new CustomerRepository();
var customers = custRep.GetCustomers();
var searchName = "Іван";
bool customerFound = false;
foreach (var cust in customers)
        {
// 1. Перша помилка
if (searchName == cust.FirstName)
            {
Console.WriteLine(
"Found: {0} {1}",
                cust.FirstName,
                cust.LastName);
                customerFound = true;
            }
        }
if (!customerFound)
        {
Console.WriteLine("Клієнта не знайдено.");
        }
Console.ReadKey();
      }
   }
}
```

Зверніть увагу, що змінна **searchName** отримала значення **"Іван"**. У циклі змінна **searchName** порівнюється з властивістю **FirstName** кожного з екземплярів класу **Customer**. Коли програма буде запущена, вона виведе наступний рядок:

```
Клієнта не знайдено.
```

У принципі, ми чекаємо, що програма повинна знайти співпадаючий рядок і вивести його на екран, але цього не відбувається. Це і є перша помилка, і зараз ми обговоримо, яким чином можна її виявити, використавши відлагоджувач VS.

3.13.6.2. Пошук помилки

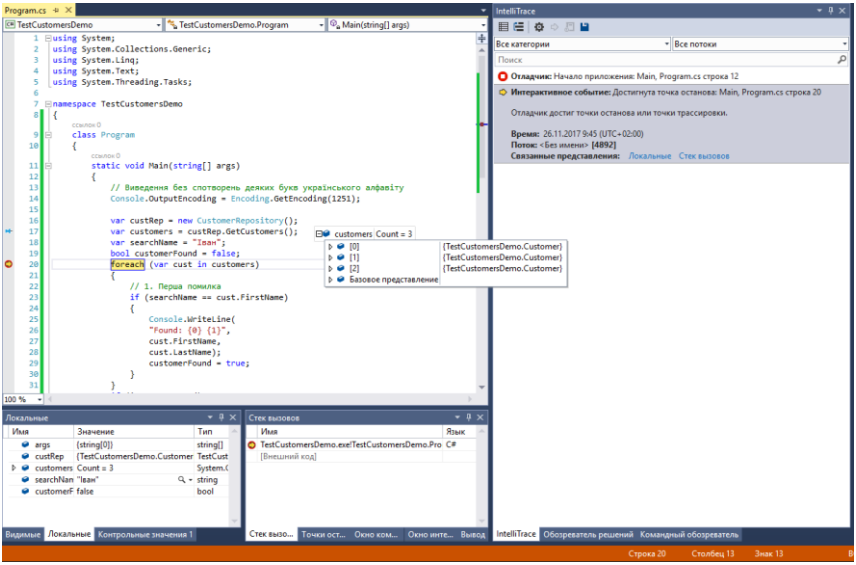
На цьому етапі ми знаємо, що в нашій програмі є помилка, і ця помилка постійно відтворюється. Це означає, що для її усунення потрібно використати відлагоджувач VS, щоб виявити джерело виниклої проблеми. У цій ситуації програма повідомляє, що запис про клієнта не знайдений, іншими словами – що у нас немає клієнта з ім'ям (**FirstName**) **Іван**. Але при цьому *ми твердо знаємо*, що у нас є такий клієнт! Тому нам треба з'ясувати, чому ж програма не може знайти потрібну інформацію. Давайте розглянемо, які кроки треба виконати за допомогою відлагоджувача VS, щоб виявити джерело помилки.

1. Розпочнемо зі встановлення точки зупину на цикл **foreach** в методі **Main**. Таке рішення не випадкове. Враховуючи природу нашої проблеми, ми вибираємо саме ту частину програми, яка з високою вірогідністю проллє світло на те, що ж стало джерелом проблеми. Подивившись на програму, нескладно прийти до висновку, що однією з причин, через яку програма не може знайти **searchName**, є те, що вона не виконує цикл **foreach**.
2. Натисніть клавішу **<F5>**, щоб запустити програму в режимі налаштування. Завдяки цьому програма не буде виконана повністю, а дійде до точки зупину, зробить паузу, і ми зможемо вивчити її стан.
3. Після того, як VS зустріне точку зупину, наведіть курсор на змінну **customers**, щоб переглянути, чи отримано нею яке-небудь значення. Ви побачите, що в списку **customers** є три значення. Той факт, що клієнти є, вказує на те, що цикл **foreach** все ж виконується, *тому припущення про його невиконання відкидається як неправильне*.
4. Далі, встановимо точку зупину на твердженні **if**, виконаємо клацання правою кнопкою миші на точці зупину і встановимо наступну умову:

```
cust.FirstName == "Іван"
```

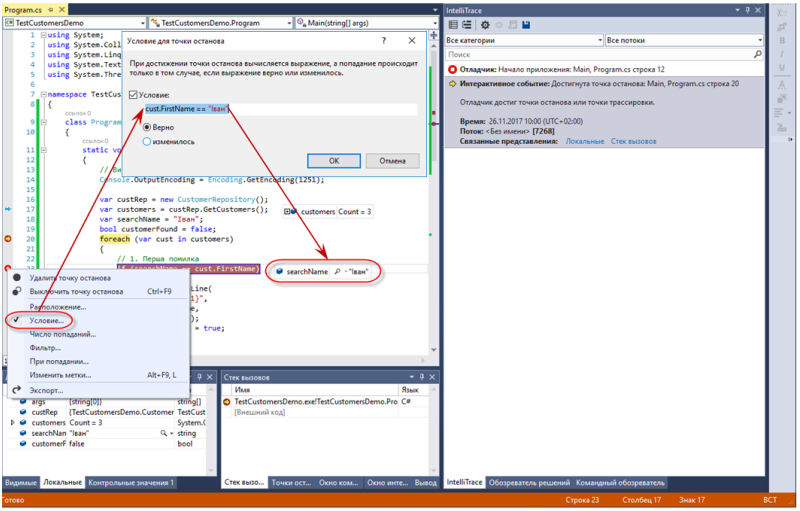
   Тут наша мета полягає в тому, щоб подивитися, що відбувається, коли твердження **if** знаходить запис, який

співпадає з **searchName**. У цей момент ми припускаємо, що серед записів про клієнтів є клієнт з ім'ям **Іван**. При роботі з невеликою програмою, ви можете використати для пошуку цього запису вікна **Autos, Locals** або **Watch**. Проте в більшості реальних практичних застосувань наявні списки, в яких містяться не два і не три записи, а набагато більше їх число. Тому замість того, щоб витрачати час на прокручування довгих списків, які налічують десятки і сотні записів, скористайтесь можливостями відлагоджувача VS *зі швидкого пошуку* потрібного запису. Майте на увазі, що навіть плани, які на перший погляд здаються раціональними і добре продуманими, працюють не завжди, як ви незабаром переконаєтеся. Але наша основна мета в даному випадку полягає в тому, щоб *зробити найпродуктивніший перший крок*. Встановлення *умовної точки зупину* демонструє, як встановлення умовних точок зупину допомагає запобігти непродуктивній витраті часу на покрокове прокручування циклів.



5. Натисніть клавішу **<F5>**, щоб знову запустити програму в режимі налаштування. Ви чекаєте зустріти точку зупину –

проте цього не відбувається. Чому? Адже ви вже знаєте, що з логікою програми все гаразд, тому що твердження **if** є звичайний оператор перевірки на рівність. Можливо, ви переглянули базу даних або будь-яке інше джерело, з якого були отримані дані, але в цьому сценарії ви абсолютно безперечно знаєте, що клієнт з ім'ям **Іван** присутній у списку. Ця ситуація ілюструє типову проблему, коли *якість даних*, з якими ви працюєте, *не відповідає потрібному рівню*.



6. Тоді давайте змінимо умову точки зупину таким чином:

```
cust.FirstName.Contains("Іван")
```

Після цього перезапустимо програму. Зверніть увагу – тепер ми підозрюємо, що наша програма працює з *некоректними даними*, тому ми і доповнюємо викликом **Contains** наш рядок, який задає умову. Ми розраховуємо знайти «зайві» символи (наприклад, пропуски) в даних, з якими виконується порівняння. Затримайте курсор над **cust.FirstName** аби подивитися на значення змінної **cust** в одному з вікон відлагоджувача, щоб переконатися, чи той

це рядок, який ми шукаємо. Тепер точка зупину спрацьовуватиме на всіх записах, які містять послідовність символів **"Іван"**, включаючи такі, як, наприклад, **Іван-Побиван**. Отже, ви отримаєте багато співпадінь, включаючи і ті записи, яких ви не шукали. Перевага полягає в тому, що у цьому випадку вам потрібно буде переглядати набагато менше число записів, і завдяки цьому ви зможете заощадити час. Якщо ви отримаєте декілька записів, ви можете натиснути клавішу **<F5>**, і точка зупину спрацьовуватиме на кожному записі, дозволяючи вам досліджувати значення. У цьому випадку, через те, що набір записів зовсім незначний, ми знайдемо потрібний запис відразу ж.

7. Натисніть клавішу **<F10>**, щоб переступити через умову **if**. У результаті ми повинні отримати відповідь на питання про те, чи правильно було оцінено умову. У цьому випадку, VS не входить у твердження **if**, а замість цього переміщається до кінця твердження, і це означає, що **searchName** і **cust.FirstName** не співпадають. У свою чергу, це говорить про те, що вам треба пильніше придивитися до **cust.FirstName**, щоб з'ясувати, що за проблема виникає з вашими даними.
8. Далі, скористаємося парочкою інструментів відлагоджувача VS, щоб дослідити значення **cust.FirstName** і виявити, чому не спрацьовує умова перевірки на рівність. Відкрийте вікно **Immediate** (або вибравши з меню команди **Debug | Windows | Immediate**, або натиснувши клавіатурну комбінацію **<CTRL>+<D>, <I>**) і виконайте оцінку наступного виразу:

```
cust.FirstName
```

У результаті ви отримаєте наступну оцінку (**рис. 3.59**):

```
"Іван "
```

Тепер вам стає очевидно, що помилка виникає у результаті помилки у даних – *на кінці імені є пропуск*. Очевидно, що **"Іван"** не співпаде з **"Іван "** *якраз через наявність зайвого*

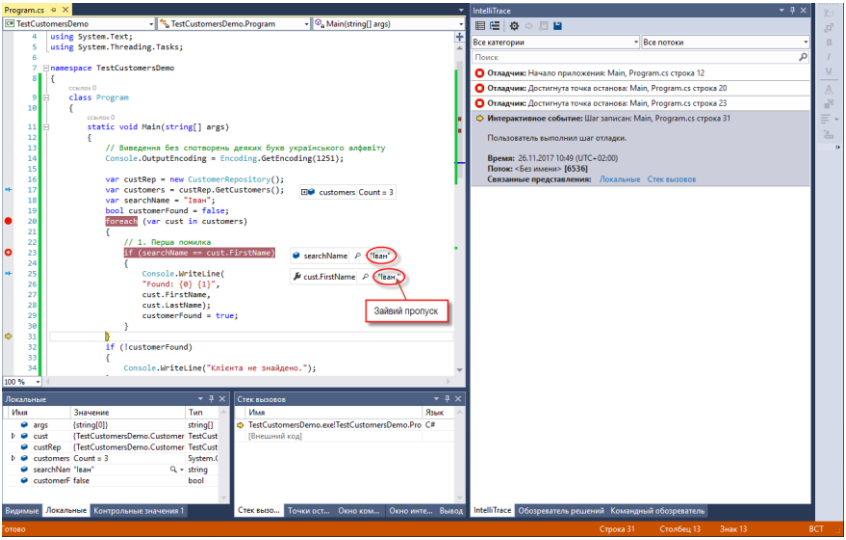
символа в даних. Окрім пропуску, зайвий символ може виявитися і яким-небудь ще недрукованим (т. т. управляючим) символом. Добре б з'ясувати, чи точно це пропуск чи якийсь інший символ. VS може допомогти і в цьому питанні.

9. Відкрийте вікно **Memory (Память)** і введіть в поле **Address** рядок **cust.FirstName**. Натисніть клавішу **<ENTER>**. У результаті ви зможете переглянути дані в шістнадцятковому представленні за тією адресою пам'яті, де вони розміщуються (**рис. 3.60**).

   На лівій межі вікна **Memory** вказуються адреси пам'яті, за якими можна виконувати їх прокручування. У цьому випадку *шістнадцятковий дамп* буде прокручений до адреси, за якою вперше зустрічається в пам'яті змінна **cust.FirstName**. У середині вікна знаходиться шістнадцяткове представлення даних, а на правій межі – представлення даних у вигляді читабельних символів, де всі символи, які не мають друкарського представлення, представлені у вигляді крапок. Як бачите, у першому рядку третього стовпця присутній рядок **"Іван"**. Шістнадцяткове представлення даних **"Іван "**. виглядає так: «**04 06 04 32 04 30 04 3d 00 20**». Звірившись з кодовою таблицею **Unicode**, ви побачите, що візуальне і шістнадцяткове представлення даних співпадають.

Як бачите, на кінці рядка (**Іван**) знаходиться байт **00 20** (див. **рис. 3.60**), а це говорить про те, що на кінці рядка **Іван** знаходиться *символ пропуску*. Деякі апаратні і програмні платформи можуть використовувати як роздільники інші, нестандартні символи, і такі символи можуть помилково потрапити в дані, що і викликає помилки, подібні до щойно розглянутих.

Примітка: Дамп пам'яті треба налаштовувати, використовуючи контекстно залежне меню. На **рис. 3.60** це відображено.



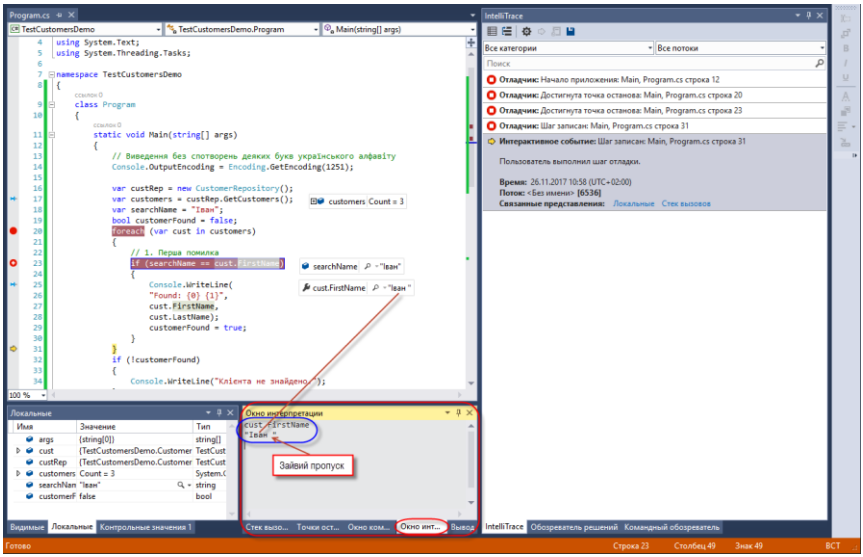


Рис. 3.59. Результат оцінки виразу **cust.FirstName**

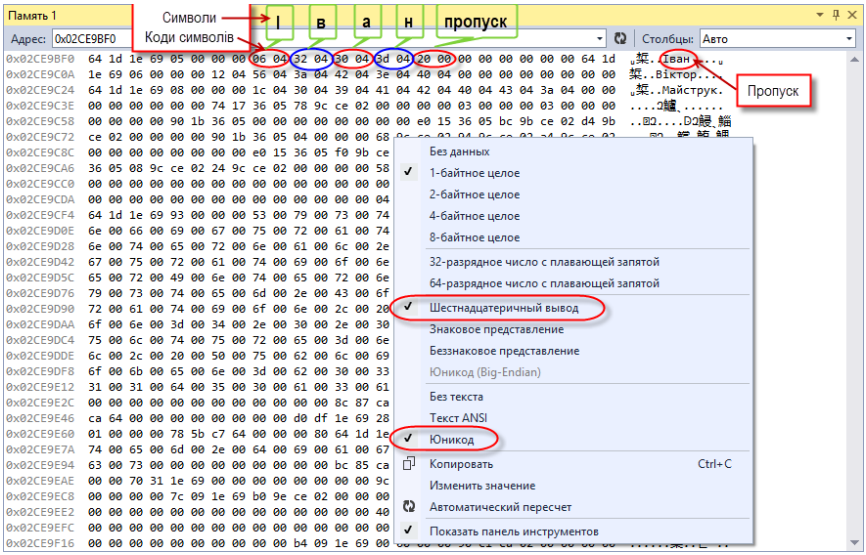


Рис. 3.60. Вміст вікна **Memory**
з відображенням проблемних даних

3.13.6.3. Виправлення першої помилки

Навіть попри те, що помилка викликана неякісними даними, а це зовсім не обов'язково може бути вашою провиною, перспективи виправлення проблеми на рівні джерела даних частенько бувають дуже ілюзорні. Це означає, що ви як програміст повинні замислитися над тим, щоб внести у ваш код такі виправлення, які не дозволять цій помилці виникати надалі. Зараз ми розробимо і застосуємо виправлення для цієї помилки. При цьому ми переслідуємо двояку мету: *по-перше*, нам потрібно проілюструвати ситуації, які часто виникають і підходи до виправлення помилок, а *по-друге*, показати, що на практиці проблеми зазвичай бувають багатогранними. Ми продемонструємо ще один, абсолютно інший приклад помилки, яка теж часто зустрічається у світі програмування. Так що процедура, описана в цьому розділі, спочатку виправить першу помилку, а потім допоможе виявити ще одну, не таку очевидну. Отже, щоб виправити першу помилку, покроково виконайте наступні операції.

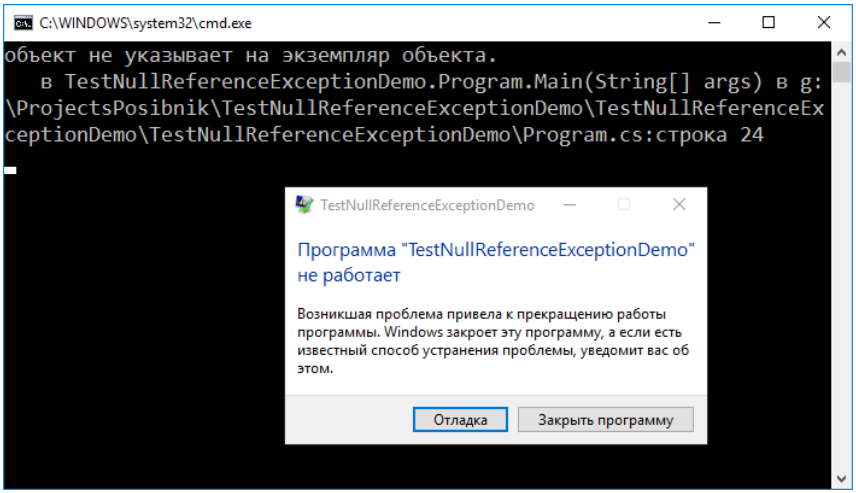
1. Натисніть клавіатурну комбінацію **<SHIFT>+<F5>**, щоб зупинити поточний сеанс налаштування.
2. Тепер реалізуємо код, який виправляє щойно виявлену нами помилку. Закоментуйте вміст циклу **foreach** і замініть його кодом, приведеним у **лістингу 3.12**. Цей код покликаний захистити програму від помилок, викликаних зайвими пропусками в даних.

Лістинг 3.12. Код, який захищає програму від зайвих пропусків у даних

```
var firstName = cust.FirstName.Trim();
var lastName = cust.LastName.Trim();
if (searchName == cust.FirstName)
{
    Console.WriteLine(
        "Знайдено: {0} {1}",
        firstName,
        lastName);
    customerFound = true;
}
```

Зверніть увагу, що наш новий код використовує метод **string.Trim** для видалення зайвих пропусків на кінці рядка, і результати такого очищення присвоюються локальним змінним. **Trim** за замовчуванням використовує символ пропуску, але перевантажується, якщо ви вкажете інший символ (на той випадок, якщо у кінці рядка виявиться інший символ, а не пропуск, див. **рис. 3.60**). Далі програма буде використовувати очищені дані.

Натисніть клавішу **<F5>**, щоб запустити програму і перевірити, як вона буде працювати. Ось тут, на жаль, вас підстерігає ще одна помилка – **NullReferenceException**. На відміну від помилок часу виконання, які є наслідком некоректних даних, у цьому VS добре допомагає у виявленні таких виключень, коли вони мають місце у вашому коді. У наступному розділі помилка **NullReferenceException** буде описана з великим числом подробиць, що повинне буде допомогти, якщо вам доведеться мати справу з усуненням подібних помилок у ваших програмах.



3.13.6.4. Налаштування і вирішення проблем NullReferenceException

Помилки типу **NullReferenceException** зустрічаються досить часто і заслуговують окремого розгляду, щоб допомогти вам ефективно боротися з такими проблемами. Як було описано у попередньому розділі (див. **крок 3**), VS зробить паузу, зустрівши помилку **NullReferenceException**. У цьому конкретному прикладі, VS призупинить виконання програми в рядку, який очищає властивість **LastName** (див. **лістинг 3.12**), повторимо ці рядки тут для зручності:

```
var firstName = cust.FirstName.Trim();
var lastName = cust.LastName.Trim();
```

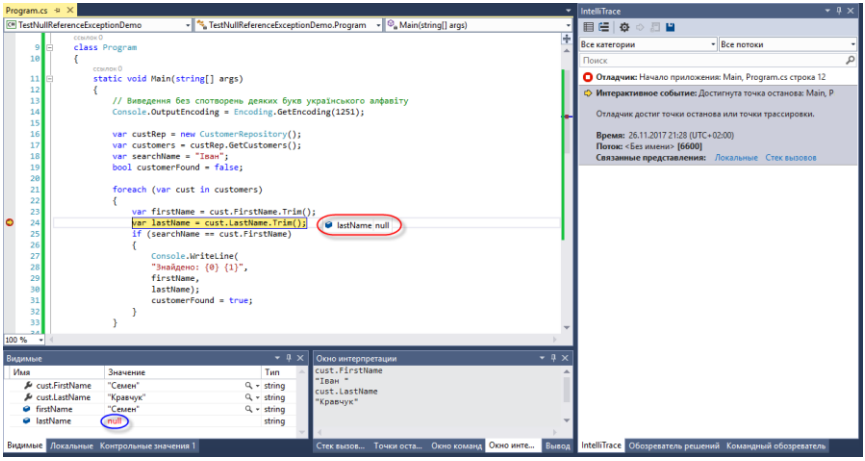
Наскільки ви пам’ятаєте, причина виклику методу **Trim** для властивостей **FirstName** і **LastName** полягала в тому, щоб виконати очищення даних від зайвих символів перед тим, як виконувати над цими даними наступні операції. Хоча основну увагу було приділено властивості **FirstName**, ми так само викликали метод **Trim** і для властивості **LastName**, щоб допомогти захистити програму від некоректних даних (хоч би вже просто – про всяк випадок). Наступні кроки покликані продемонструвати, як треба використати VS для

аналізу поточної ситуації і приймати рішення про виправлення помилки.

1. Запустіть середовище розробки VS, відкрийте проект і запустіть програму. Дочекайтеся, коли VS зупинить виконання, викинувши помилку **NullReferenceException**.
2. Затримайте курсор над властивістю **cust.LastName**, щоб переглянути значення. Як альтернативний варіант, ви можете переглянути значення в одному з вікон, доступних в режимі налаштування. Зверніть увагу, що значення **LastName** встановлене на **null**. Це і є критична точка нашого аналізу – ми виявили значення, встановлене на **null**. Очевидно, що значення **cust** не дорівнює **null**, тому що попереднє твердження, яке очікує дані **FirstName**, було виконано без помилки, що і підтверджується шляхом вивчення змінної **firstName**. У цьому прикладі знайти значення **null** було зовсім не складно, тому що воно з'являється в тому рядку, на якому VS робить паузу. Бувають і складніші ситуації, в яких ви передаєте об'єкт методу з бібліотеки від сторонніх розробників, і при цьому у вас немає початкового коду цієї бібліотеки, причому VS зупиняється на рядку, в якому здійснюється виклик методу. У цьому випадку, вам потрібно буде досліджувати значення, яке передаються методу, щоб з'ясувати, яке ж з них дорівнює **null**.

   Щойно ви знайдете значення **null**, ви зрозумієте, чому виникла помилка **NullReferenceException**. Значення **null** фактично означає відсутність значення – а саме, що змінній взагалі не було присвоєно ніякого значення. Якщо ви спробуєте послатися на змінну зі значенням **null**, ви отримаєте повідомлення про помилку **NullReferenceException**. Це має сенс, бо ви намагаєтеся виконати операцію над змінною, яка не визначена. У цьому конкретному прикладі, значення **LastName** дорівнює **null**, але ми все одно посилаємося на **LastName**, викликаючи метод **Trim**. Це нелогічно, тому що немає самого рядка, з якого можна було б видаляти зайві символи – сама рядкова змінна встановлена на **null**.

Помилка **NullReferenceException** виникає, для захисту програми від виконання некоректної операції. Знайшовши значення **null** і з'ясувавши причину появи повідомлення про помилку, треба з'ясувати, чому ж змінна набуває значення **null**, щоб прийняти правильне рішення про те, як же виправити цю проблему.



3. У вікні **Immediate** введіть наступну команду:

```
customers.IndexOf(cust)
```

Ця команда поверне значення **1**, яке є індексом поточного запису типу **Customer.cust**, у списку клієнтів. Ця інформація дозволить досягти істотного заощадження часу при спробі знайти цей об'єкт серед інших даних програми.

4. На даний момент відлагоджувач зробив паузу на рядку, який виконує очищення **LastName**, в якій і сталася помилка **NullReferenceException**, і на лівій межі є жовта стрілка, яка символізує точку зупину (див. попередній рисунок). З допомогою миші перетягніть її до рядка, з якого здійснюється виклик **GetCustomers**. Зараз наше завдання полягає в тому, щоб з'ясувати, де і в який момент змінна отримує значення **null**. При удачі, ми можемо зробити

зупинку в цій точці і, можливо, з'ясувати причину помилки, через яку змінна набуває значення **null**.

5. Натисніть клавішу **<F11>**, щоб увійти до методу **GetCustomers**. VS перейде до першого рядка методу **GetCustomer**.
6. Двічі натисніть клавішу **<F10>**, щоб побачити, які значення повернуті. Цей приклад настільки простий, що ви можете виявити дані візуально. Проте, в реальних сценаріях, вам, можливо, доведеться запускати такий код, який виконує запити до баз даних або отримує їх з інших джерел, і готувати ці дані в такій формі, щоб їх можна було передавати будь-якому викликаючому коду. Робота з базами даних вимагає окремого розгляду. У цьому ж прикладі основна мета – розповісти про техніку налаштування, не відволікаючи уваги на речі, які не мають прямого відношення до справи. Тому нам треба обстежувати дані і подивитися, чи не є вони джерелом значень типу **null**. Для цієї мети введемо у вікно **Immediate** наступну команду:

```
customers[1].LastName
```

Додатково вам треба переглянути список клієнтів в одному з вікон налаштування: **Autos, Locals** або **Watch**, звернувши особливу увагу на об'єкт колекції **Customer** з індексом, рівним **1**. Згадаємо, що раніше на **кроці 3** ми вже з'ясували, що об'єкт типу **Customer**, який цікавить нас, має індекс **1**. На підставі отриманого результату нам вдалося з'ясувати, що властивість **LastName** для цього об'єкта типу **Customer** було встановлено на **null** у джерелі даних, і ми нічого не можемо зробити, щоб воно набувало інших значень, а не **null**. Тут ми стикаємося з ще одним різновидом некоректних даних. Якщо ви можете простежити чітку тенденцію, ви будете цілком праві, коли станете з недовірою відноситися до даних, які постачаються конкретним користувачем або отримуються з тієї бази даних, звідки ви часто отримуєте таку некоректну інформацію. На цьому етапі, ми отримали всю інформацію, яка нам потрібна для виправлення

помилки (див. **рис. 3.61**). Це виправлення гарантує, що ми ніколи, не передаватимемо методам дані типу **null**. Натисніть клавіатурну комбінацію **<SHIFT>+<F5>**, щоб завершити сеанс налаштування.

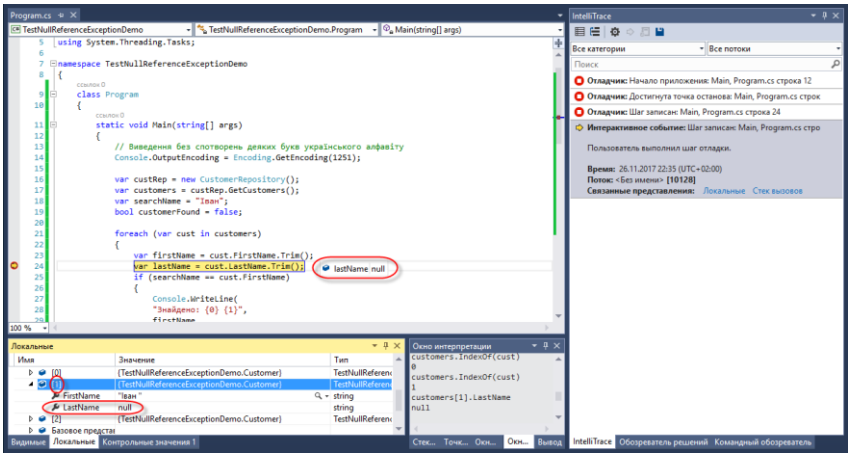


Рис. 3.61. Властивість **LastName** встановлено на **null** у джерелі даних

7. У цьому прикладі ми виправимо проблему за рахунок того, що реалізуємо перевірку значень **null** перед тим, як передавати ці дані методам, які викликаються. У випадку якщо яка-небудь змінна виявиться рівною **null**, ми будемо заміняти її стандартним значенням за замовчуванням. Закоментуйте вміст циклу **foreach** і замініть його кодом, приведеним у **лістингу 3.13**.

Лістинг 3.13. Код, який виконує перевірку даних на рівність null

```
string firstName = string Empty;
if (cust.FirstName != null)
{
      firstName = cust.FirstName.Trim();
}
string lastName =
      cust.LastName == null ?
```

```
        "" : cust.LastName.Trim();
if (searchName == firstName)
{
        Console.WriteLine(
            "Знайдено: {0} {1}",
            firstName,
            lastName);
        customerFound = true;
}
```

Остаточний текст програми

Лістинг 3.14. Код, який виконує перевірку даних на рівність null

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

namespace TestNullReferenceExceptionDemo
{
        class Program
        {
            static void Main(string[] args)
            {
            // Виведення без спотворень деяких букв українського алфавіту
                        Console.OutputEncoding = Encoding.GetEncoding(1251);

                        var custRep = new CustomerRepository();
                        var customers = custRep.GetCustomers();
                        var searchName = "Іван";
                        bool customerFound = false;

                        foreach (var cust in customers)
                        {
                                    string firstName = string Empty;
                                    if (cust.FirstName != null)
                                    {
```

```
                    firstName = cust.FirstName.Trim();
                }
                string lastName =
                    cust.LastName == null ?
                    "" : cust.LastName.Trim();
                if (searchName == firstName)
                {
                    Console.WriteLine(
                        "Знайдено: {0} {1}",
                        firstName,
                        lastName);
                        customerFound = true;
                }
            }
            if (!customerFound)
            {
                Console.WriteLine("Клієнта не знайдено.");
            }
            Console.ReadKey();
        }
    }
    public class Customer
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }

    public class CustomerRepository
    {
        public List <Customer> GetCustomers()
        {
            var customers = new List<Customer>
            {
                new Customer
                {
                    FirstName = "Семен",
                    LastName = "Кравчук"
                },
```

```
                    new Customer
                    {
                        FirstName = "Іван "
                    },
                    new Customer
                    {
                        FirstName = "Віктор",
                        LastName = "Майструк"
                    }
                };
                return customers;
            }
        }
}
```

Код, приведений в **лістингу 3.13**, виправляє проблему двома різними способами, даючи вам можливість скористатися різними підходами, залежно від того стилю, якому ви віддаєте перевагу. Спочатку виконується перевірка властивостей **FirstName** і **LastName**, щоб з'ясувати, чи не отримує одне з них значення **null**. Якщо ці властивості не дорівнюють **null**, то це означає, що вони містять прийнятні рядкові значення, і з ними цілком можна працювати. Інакше, ми повертаємо порожній рядок.

Виведення порожнього рядка як значення за замовчуванням характерна тільки для цього прикладу. На практиці вам доведеться орієнтуватися по ситуації і вирішувати, чи є сенс присвоювати значення за замовчуванням. Наприклад, присутність значення **null** може відповідати неправильній умові, і ви можете зареєструвати таку подію в журналі і не давати користувачеві можливості продовжувати виконання поточної операції. Ще один варіант дій полягає в тому, щоб пропустити поточний запис, обробити всі інші, а потім вивести список записів, які не були оброблені. Ви можете вибрати будь-який із запропонованих тут підходів або відкинути їх всі. Отже вам треба замислитися про те, що в кожній конкретній ситуації означає робота зі значенням **null**, а не просто усунути проблему нульових покажчиків.

8. Натисніть клавішу **<F5>**, щоб запустити програму. Ви отримаєте наступний результат:

```
Знайдено: Іван
```

Підсумок

Після опрацювання цього матеріалу ви повинні набути базових навичок налаштування коду. Розділ «Інтегроване середовище розробки Microsoft Visual Studio .NET» детально описує, як вивчати структуру вашого коду в ході розробки. Далі було розказано про те, як треба встановлювати точки зупину і виконувати їх індивідуальне налаштування. Потім була розглянута методика покрокового виконання коду в режимі налаштування, навігація по застосуванню, вхід в методи і вихід з них, а також зміна каталогу, в якому розташовується виконуваний файл вашого застосування. Нарешті, були розглянуті вікна налаштування, які дозволяють вивчити статус застосування у кожен конкретний момент часу. Зокрема, було описано використання вікна історії налаштування та його застосування для вивчення стану застосування на кожному з етапів сеансу налаштування. Нарешті, у завершальному розділі використання усіх цих методик було проілюстровано на практичному прикладі, який показує, як усувати помилки у коді застосувань за допомогою відлагоджувача VS.

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Дейтел П. Как программировать на Visual C# 2012. / П. Дейтел, Х. Дейтел. – СПб.: Питер, 2014. – 864 с.: ил. – (серия «Библиотека программиста»).
2. Дейтел Х. C#. / Х. Дейтел, П. Дейтел, Дж. Листфилд, Т. Нието, Ш. Йегер, М. Златкина – СПб.: БХВ-Петербург, 2006. – 1056 с.
3. Макконнелл С. Совершенный код. Мастер класс. / С. Макконнелл – М.: Издательство «Русская редакция», 2010. – 896 с.
4. Нейгел К. C# 4.0 и платформа .NET 4 для профессионалов. / К. Нейгел, Б. Ивьен, Д. Глинн, К. Уотсон – М.: ООО «И.Д. Вильямс», 2011. – 1440 с.
5. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. / Дж. Рихтер – СПб.: Питер, 2013. – 896 с.
6. Роббинс Дж. Отладка приложений для Microsoft .NET и Microsoft Windows. / Дж. Роббинс – М.: Издательство «Русская редакция», 2004. – 736 с.
7. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. / Э. Троелсен – М.: ООО «И.Д. Вильяме», 2011. – 1392 с.
8. Шилдт Г. C# 4.0: полное руководство. / Г. Шилдт – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.