МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ

Кафедра інформаційних технологій проектування в машинобудуванні

КОНСПЕКТ ЛЕКЦІЙ
З ДИСЦИПЛІНИ
«АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ»

Для студентів інституту промислових технологій, дизайну та менеджменту

Перший (бакалаврський) рівень вищої освіти
Спеціальність – 122 КОМП'ЮТЕРНІ НАУКИ
Спеціалізація – *Комп'ютерний дизайн*

І курс, 1, 2 семестри

Затверджено на засіданні
кафедри інформаційних технологій
проектування в машинобудуванні
Протокол №1 від 29.08.17 р.

Одеса – 2017

Конспект лекцій з дисципліни «Алгоритмізація та програмування» для студентів спеціальності 122 Комп’ютерні науки, спеціалізація — Комп’ютерний дизайн / Укл.: *А.В. Павлишко, О.В. Савєльєва.* Одеса: ОНПУ, 2017. – 123 с.

Укладачі: **Павлишко А.В.,** канд. техн. наук, доц.
Савєльєва О.В., канд. техн. наук, доц.

В даному конспекті лекцій викладені основи алгоритмізації та програмування, принципів створення алгоритмів — шляхів вирішення різноманітних завдань і перекладання їх на обчислювальні ресурси. Розглядаються основні принципи побудови алгоритмів, а також основи мови програмування високого рівня C# для їх реалізації.

Конспект лекцій включає теми, що охоплюють основні конструкції мови C#. Необхідним доповненням до даної роботи є лабораторний практикум, при виконанні завдань якого студенти отримують навички алгоритмізації та програмування, освоюють способи вирішення, в першу чергу, обчислювальних задач на персональному комп'ютері

Конспект лекцій призначений для студентів денної форми навчання Одеського національного політехнічного університету спеціальності 122 «Комп'ютерні науки» спеціалізації «Комп'ютерний дизайн», але може бути використаний і іншими студентами які вивчають курс «Алгоритмізація та програмування» або мову програмування C#.

ЗМІСТ

ВСТУП

При викладанні даного матеріалу передбачається, що студенти, хоча б у загальних рисах, знають принципи роботи персональних комп'ютерів. Також вважається, що студенти володіють відомостями про те, що таке програмне забезпечення ПК і функціонуванні операційних систем. У конспекті наводяться основні відомості, необхідні інженеру для вирішення своїх прикладних задач на основі процедурної частини мови програмування високого рівня C#. Розглядаються види та склад систем програмування, види і призначення різних мов програмування. Наводяться правила побудови програм, використання математичних функцій і виразів для обчислення значень за складними формулами в лінійних алгоритмах. Аналізуються способи записи різних видів розгалужуються і циклічних алгоритмів, обробки одновимірних і двовимірних масивів, а також строковий інформації. Приділяється багато уваги способам структуризації програм для зменшення обсягу програмного коду і зручності читання тексту програми.

Протягом всього курсу проводиться ідея первинності побудови алгоритму розв'язання задачі і вторинності написання програмного коду на мові високого рівня. Такий підхід пов'язаний не тільки з тим, що саме так і відбувається в інженерній практиці, але і з студентів скоріше написати програму, не продумавши і не сконструювавши в будь-якому вигляді алгоритм вирішення задачі. У зв'язку з цим приділяється велика увага формалізації процесу переходу від алгоритму рішення, записаного в графічній формі, до алгоритму, записаному на мові програмування.

Дуже багато інженерних завдань можна вирішити за допомогою вже готових програм — прикладних пакетів. Вони вміють інтегрувати і диференціювати, вирішувати системи лінійних і нелінійних рівнянь тощо.

Однак не завжди можна пристосувати своє завдання під можливості пакета, часто буває простіше і швидше написати свою програму. Навчитися це робити дуже важливо і тому, що програмування допомагає розвинути логічне мислення і сформувати навички структуризації, формалізації процесу вирішення будь-якої задачі.

Розглянемо етапи рішення фізичної задачі за допомогою персонального комп'ютера і з'ясуємо, яке місце займає безпосередньо програмування в цьому процесі.

Фізична постановка задачі. Тут відбувалася регулярна подача фізичної моделі процесу або явища, тобто формулювання припущень, що відображають наші уявлення про нього.

Математична постановка задачі. На цьому етапі на основі фізичного формулювання завдання вибираються змінні, що підлягають визначенню, записуються обмеження, зв'язки між змінними, в сукупності утворюють математичну модель розв'язуваної проблеми. В результаті інженерне завдання набуває вигляду формалізованої математичної задачі, записаної у вигляді рівнянь.

Перш за все, слід нагадати, що вивчення мови програмування являє собою знайомство з формальними правилами запису алгоритмів для їх подальшого виконання комп'ютером. Формальність виникає з самих принципів, закладених в архітектуру обчислювальних пристроїв, і жорсткості математичної логіки. Тому, постарайтеся сприйняти всі досить суворі правила як неминучість, налаштувати себе на серйозну, скрупульозну, часом складну роботу. Однак не варто боятися, засмучуватися і нарікати на долю: трохи акуратності, уваги, знання матеріалу — і ви вже програміст.

І курс, 1 семестр
Семестровий модуль 1.

ЛЕКЦІЯ 1.
Архітектура фон Неймана

Архітектура фон Неймана (англ. Von Neumann architecture) - широко відомий принцип спільного зберігання програм і даних в пам'яті комп'ютера. Обчислювальні системи такого роду часто позначають терміном «машина фон Неймана», проте, відповідність цих понять не завжди однозначно. У загальному випадку, коли говорять про архітектуру фон Неймана, мають на увазі фізичне відділення процесорного модуля від пристроїв зберігання програм і даних.
Наявність заданого набору виконуваних команд і програм було характерною рисою перших комп'ютерних систем. Сьогодні подібний дизайн застосовують з метою спрощення конструкції обчислювального пристрою. Так, настільні калькулятори, в принципі, є пристроями з фіксованим набором виконуваних програм. Їх можна використовувати для математичних розрахунків, але неможливо застосувати для обробки тексту і комп'ютерних ігор, для перегляду графічних зображень або відео. Зміна вбудованої програми для такого роду пристроїв вимагає практично повної їх переробки, і в більшості випадків неможливо. Втім, перепрограмування ранніх комп'ютерних систем все-таки виконувалося, проте вимагало величезного обсягу ручної роботи з підготовки нової документації, перекоммутации і перебудови блоків і пристроїв і т. П. Все змінила ідея збереження комп'ютерних програм в загальній пам'яті. На час її появи використання архітектур, заснованих на наборах виконуваних інструкцій, і уявлення обчислювального процесу як процесу виконання інструкцій, записаних в програмі, надзвичайно збільшило гнучкість обчислювальних систем в плані обробки даних. Один і той же підхід до розгляду даних і інструкцій зробив легкої завдання зміни самих програм.

Принципи фон Неймана

У 1946 році троє вчених - Артур Беркс, Герман Голдстайн і Джон фон Нейман - опублікували статтю «Попередній розгляд логічного конструювання електронного обчислювального пристрою». У статті обгрунтовувалося використання двійкової системи для представлення даних в ЕОМ (переважно для технічної реалізації, простота виконання арифметичних і логічних операцій - до цього машини зберігали дані в десятковому вигляді), висувалася ідея використання загальної пам'яті для програми і даних. Ім'я фон Неймана було досить широко відомо в науці того часу, що відсунуло на другий план його співавторів, і дані ідеї отримали назву «принципи фон Неймана».

1. Принцип двійкового кодування. Згідно з цим принципом, вся інформація, яка надходить в ЕОМ, кодується за допомогою двійкових сигналів (двозначних цифр, що бітів) і розділяється на одиниці, звані словами.

2. Принцип однорідності пам'яті. Програми та дані зберігаються в одній і тій же пам'яті. Тому ЕОМ не розрізняє, що зберігається в даній комірці пам'яті - число, текст або команда. Над командами можна виконувати такі ж дії, як і над даними.

3. Принцип адресується пам'яті. Структурно основна пам'ять складається з пронумерованих осередків; процесору в довільний момент часу доступна будь-яка ячейка.Отсюда слід можливість давати імена областям пам'яті, так, щоб до зберігаються в них значень можна було б згодом звертатися або міняти їх в процесі виконання програми з використанням привласнених імен.

4. Принцип послідовного програмного управління. передбачає, що програма складається з

набору команд, які виконуються процесором автоматично один за одним в певній послідовності.

5. Принцип жорсткості архітектури. Незмінюваність в процесі роботи топології, архітектури, списку команд.

Комп'ютери, побудовані на цих принципах, відносять до типу Фоннеймановских.

Система числення

Система числення:
• дає уявлення безлічі чисел (цілих і / або речових);
• дає кожному числу унікальну виставу (або, принаймні, стандартне уявлення);
• відображає алгебраїчну і арифметичну структуру чисел.
Системи числення поділяються на позиційні, непозиційної і змішані.
Чим більше підставу системи числення, тим менша кількість розрядів (тобто записуються цифр) потрібний час запису числа в позиційних системах числення.

Позиційні системи числення

У позиційних системах числення один і той же числовий знак (цифра) в запису числа має різні значення в залежності від того місця (розряду), де він розташований. Винахід позиційної нумерації, заснованої на помісному значенні цифр, приписується шумерам і вавилоняни; розвинена була така нумерація індусами і мала неоціненні наслідки в історії людської цивілізації. До числа таких систем відноситься сучасна десяткова система числення, виникнення якої пов'язано з рахунком на пальцях. У середньовічній Європі вона з'явилася через італійських купців, в свою чергу запозичили її у мусульман.
Під позиційною системою числення зазвичай розуміється b-річної системі числення, яка визначається цілим числом b> 1, званим підставою системи числення. Ціле число x в b-річної системі числення представляється у вигляді кінцевої лінійної комбінації ступенів числа b:

$x = \sum_{k=0}^{n-1} a_k b^k$, де a_k — це цілі числа, звані цифрами, що задовольняють нерівності $0 \leq a_k \leq (b-1)$.

Кожна ступінь b^k в такому записі називається ваговим коефіцієнтом розряду. Старшинство розрядів і відповідних їм чисел визначається значенням показника k (номером розряду). Зазвичай для ненульового числа x вимагають, щоб старша цифра an - 1 в його b-ковий поданні була також ненульовий.
Якщо не виникає різночитань (наприклад, коли всі цифри видаються у вигляді унікальних письмових знаків), число x записують у вигляді послідовності його b-ковий цифр, що перераховуються за зменшенням старшинства розрядів зліва направо:
$x = a_{n-1}a_{n-2}\ldots a_0.$

Наприклад, число сто три представляється в десятковій системі числення у вигляді:
$103 = 1 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0.$

Найбільш вживаними в даний час позиційними системами є:
• 1 - одинична (рахунок на пальцях, зарубки, вузлики «на пам'ять» та ін.);
• 2 - двійкова (у дискретної математики, інформатики, програмування);
• 3 - потрійна;
• 8 - восьмерична;
• 10 - десяткова (використовується повсюдно);
• 12 - двенадцатирічня (рахунок дюжинами);
• 16 - шістнадцяткова (використовується в програмуванні, інформатики);
• 60 - шістдесяткова (одиниці вимірювання часу, вимірювання кутів і, зокрема, координат, довготи і широти).Смешанные системы счисления

Змішана система числення є узагальненням b-річної системи числення і також часто відноситься до позиційних систем числення. Підставою змішаної системи числення є зростаюча послідовність чисел $\{b_k\}_{k=0}^{\infty}$, і кожне число x в ній представляється як лінійна комбінація:

$$x = \sum_{k=0}^{n-1} a_k b_k$$

, де на коефіцієнти a_k, звані як і раніше цифрами, накладаються деякі обмеження. Записом числа x в змішаній системі числення називається перерахування його цифр в порядку зменшення індексу k, починаючи з першого ненульового.
Залежно від виду b_k як функції від k змішані системи числення можуть бути статечними, показовими і т. П. Коли $b_k = b^k$ для деякого b, змішана система числення збігається з b-річної системою числення.
Найбільш відомим прикладом змішаної системи числення є представлення часу у вигляді кількості діб, годин, хвилин і секунд. При цьому величина «d днів, h годин, m хвилин, s секунд» відповідає значенню
$d \cdot 24 \cdot 60 \cdot 60 + h \cdot 60 \cdot 60 + m \cdot 60 + s$секунд.

Факторіальна система числення

У факториальной системі числення підставами є послідовність факториалов $b_k = k!$, і кожне натуральне число x представляється у вигляді:

$$x = \sum_{k=1}^{n} d_k k!$$

, где $0 \le d_k \le k$.

Фібоначчієва система числення

Фібоначчієва система числення грунтується на числах Фібоначчі. Кожне натуральне число x в ній представляється у вигляді:

$$x = \sum_{k=0}^{n} f_k F_k$$

, де F_k — числа Фібоначчі, $f_k \in \{0, 1\}$, при цьому в записи $f_n f_{n-1} \cdots f_0$

не зустрічається дві одиниці підряд.

Непозиційні системи числення

У непозиційних системах числення величина, яку позначає цифра, не залежить від положення в числі. При цьому система може накладати обмеження на положення цифр, наприклад, щоб вони були розташовані в порядку убування.

Біноміальна система числення

Подання, що використовує біноміальні коефіцієнти

$$x = \sum_{k=1}^{n} \binom{c_k}{k}$$

, где $0 \le c_1 < c_2 < \ldots < c_n$.

Система залишкових класів (СОК)

Подання числа в системі залишкових класів засноване на понятті вирахування і китайської теореми про залишки. СОК визначається набором взаємно простих модулів $(m_1, m_2, \ldots, m_n)$ з множенням $M = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ так, що кожному цілому числу x з відрізка [0, M - 1] ставиться у відповідність набір відрахувань $(x_1, x_2, \ldots, x_n)$, де

$$x \equiv x_1 \pmod{m_1};$$
$$x \equiv x_2 \pmod{m_2};$$
$$\ldots$$
$$x \equiv x_n \pmod{m_n}.$$

При цьому китайська теорема про залишки гарантує однозначність подання для чисел з відрізка [0, M - 1].
В СОК арифметичні операції (додавання, віднімання, множення, ділення) виконуються покомпонентно, якщо про результат відомо, що він є цілочисельним і також лежить в [0, M - 1].

Недоліками СОК є можливість подання лише обмеженої кількості чисел, а також відсутність ефективних алгоритмів для порівняння чисел, представлених в СОК. Порівняння зазвичай здійснюється через переклад аргументів з СОК в змішану систему числення з підстав $(m_1, m_1 \cdot m_2, \ldots, m_1 \cdot m_2 \cdot \ldots \cdot m_{n-1})$.

Система числення Штерна-Броко

Система числення Штерна-Броко - спосіб запису позитивних раціональних чисел, заснований на дереві Штерна-Броко.

Системи числення різних народів

Давньоєгипетська система числення

Давньоєгипетська десяткова непозиційних система числення виникла в другій половині третього тисячоліття до н. е. Для позначення чисел 0, 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7 використовувалися спеціальні цифри. Числа в єгипетській системі числення записувалися як комбінації цих цифр, в яких кожна з цифр повторювалася не більше дев'яти раз. Значення числа одно простої суми значень цифр, які беруть участь в його записи.

Алфавітні системи числення

Алфавітними системами числення користувалися стародавні вірмени, грузини, греки (іонічна система числення), араби (абджадія), євреї (див. Гематрія) та інші народи Близького Сходу. У слов'янських богослужбових книгах грецька алфавітна система була переведена на літери кирилиці.
Єврейська система числення
Єврейська система числення в якості цифр використовує 22 літери єврейського алфавіту. Кожна буква має своє числове значення від 1 до 400. Нуль відсутня. Цифри, записані таким чином, найбільш часто можна зустріти в нумерації років за іудейським календарем.
Римська система числення
Канонічним прикладом майже непозиционной системи числення є римська, в якій в якості цифр використовуються латинські літери:

I позначає 1,

V - 5,
X - 10,
L - 50,
C - 100,
D - 500,
M - 1000
Наприклад, II = 1 + 1 = 2
тут символ I позначає 1 незалежно від місця в числі.
Насправді, римська система не є повністю непозиционной, так як менша цифра, що йде перед більшою, віднімається з неї, наприклад:
IV = 4, в той час як:
VI = 6

Система числення майя

Майя використовували 20-річної систему числення за одним винятком: у другому розряді було не 20, а 18 ступенів, тобто за числом (17) (19) відразу слід було число (1) (0) (0). Це було зроблено для полегшення розрахунків календарного циклу, оскільки (1) (0) (0) = 360 приблизно дорівнює числу днів в сонячному році. Для запису основними знаками були точки (одиниці) і відрізки (п'ятірки).
стос інків
Прообразом баз даних, широко використовувалися в Центральних Андах (Перу, Болівія) у державних та громадських цілях в I-II тисячолітті н.е., була вузликова писемність Інків - стос, що складалася як з числових записів десяткової системи, так і не числових записів в двійковій системі кодування. У стос застосовувалися первинні та додаткові ключі, позиційні числа, кодування кольором та освіта серій повторюваних даних. Стос вперше в історії людства використовувалося для застосування такого способу ведення бухгалтерського обліку як подвійний запис

Переклад чисел з однієї системи числення в іншу

Переклад чисел з шістнадцятковій системи в десяткову

Для перекладу шістнадцятирічного числа в десяткове необхідно це число представити у вигляді суми добутків ступенів підстави шістнадцятковій системи числення на відповідні цифри в розрядах шістнадцятирічного числа.
Наприклад, потрібно перевести шістнадцяткове число 5A3 в десяткове. У цьому числі 3 цифри. Відповідно до вищезазначеного правилом представимо його у вигляді суми ступенів з основою 16:
$5A3_{16} = 3 \cdot 160 + 10 \cdot 161 + 5 \cdot 162 = 3 \cdot 1 + 10 \cdot 16 + 5 \cdot 256 = 3 + 160 + 1280 = 144310$

Переклад чисел з двійкової системи в шістнадцяткову і навпаки

Для перекладу багатозначного двійкового числа в шістнадцяткову систему потрібно розбити його на тетради справа наліво і замінити кожну тетраду відповідної шістнадцятковій цифрою. Для перекладу числа з шістнадцятковій системи в двійкову потрібно замінити кожну його цифру на відповідну тетраду. Наприклад:

$0101101000112 = 0101\ 1010\ 0011 = 5A3_{16}$

ЛЕКЦІЯ 2.
АЛГОРИТМИ

Алгоритм, від імені вченого аль-Хорезмі - точний набір інструкцій, що описують порядок дій виконавця для досягнення результату рішення задачі за кінцевий час. У старій трактуванні замість слова «порядок» використовувалося слово «послідовність», але в міру розвитку паралельності в роботі комп'ютерів слово «послідовність» стали замінювати більш загальним словом «порядок». Це пов'язано з тим, що робота якихось інструкцій алгоритму може бути залежна від інших інструкцій або результатів їх роботи. Таким чином, деякі інструкції повинні виконуватися строго після завершення роботи інструкцій, від яких вони залежать. Незалежні інструкції або інструкції, що стали незалежними через завершення роботи інструкцій, від яких вони залежать, можуть виконуватися в довільному порядку, паралельно або одночасно, якщо це дозволяють використовувані процесор і операційна система.
Раніше часто писали «алгорифм», зараз таке написання використовується рідко, але, тим не менше, має місце (наприклад, Нормальний алгорифм Маркова).

Часто в якості виконавця виступає деякий механізм (комп'ютер, токарний верстат, швейна машина), але поняття алгоритму необов'язково відноситься до комп'ютерних програм, так, наприклад, чітко описаний рецепт приготування страви також є алгоритмом, в такому випадку виконавцем є людина.
Поняття алгоритму належить до первісних, основним, базисним поняттям математики. Обчислювальні процеси алгоритмічного характеру (арифметичні дії над цілими числами, знаходження найбільшого загального дільника двох чисел і т. Д.) Відомі людству з глибокої давнини. Однак, в явному вигляді поняття алгоритму сформувалося лише на початку XX століття.

Часткова формалізація поняття алгоритму почалася зі спроб вирішення проблеми дозволу, яку сформулював Давид Гільберт в 1928 році. Наступні етапи формалізації були необхідні для визначення ефективних обчислень або «ефективного методу»; серед таких формалізацій - рекурсивні функції Геделя - Ербрана-Кліні 1930, 1934 і 1935 рр., λ-числення Алонзо Черча 1936 р, «Формулювання 1» Еміля Поста 1936 року і машина Тьюринга. У методології алгоритм є базисним поняттям і отримує якісно нове поняття як оптимальності в міру наближення до прогнозованого абсолюту. У сучасному світі алгоритм в формалізованому виразі складає основу освіти на прикладах, за подобою. На основі подібності алгоритмів різних сфер діяльності була сформована концепція (теорія) експертних систем.
Сучасне формальне визначення алгоритму було дано в 30-50-х роках XX століття в роботах Тьюринга, Посту, Черча (теза Черча - Тьюринга), Н. Вінера, А. А. Маркова.

Саме слово «алгоритм» походить від імені перського вченого Абу Абдуллах Мухаммеда ібн Муса аль-Хорезмі (алгоритм - аль-Хорезмі). Близько 825 року він написав твір, в якому вперше дав опис придуманої в Індії позиційної десяткової системи числення. На жаль, перський оригінал книги не зберігся. Аль-Хорезмі сформулював правила обчислень в новій системі і, ймовірно, вперше використав цифру 0 для позначення пропущеної позиції в записі числа (її індійське назва араби перевели як as-sifr або просто sifr, звідси такі слова, як «цифра» і «шифр»). Приблизно в цей же час індійські цифри почали застосовувати і інші арабські вчені. У першій половині XII століття книга аль-Хорезмі в латинському перекладі проникла в Європу. Перекладач, ім'я якого до нас не дійшло, дав їй назву Algoritmi de numero Indorum («Алгоритми про рахунок індійському»). По-арабськи ж книга називалася («Книга про складання і віднімання»). З оригінальної назви книги відбувається слово Алгебра (алгебра - аль-джебр - додавання).
Алгоритм - це мистецтво рахунку за допомогою цифр, але спочатку слово «цифра» відносилося лише до нуля. Знаменитий французький трувер Готьє де Куанса (Gautier de Coincy, 1177-1236) в одному з віршів використовував слова algorismus-cipher (які означали цифру 0) як метафору для характеристики абсолютно нікчемного людини. Очевидно, розуміння такого способу вимагало відповідної підготовки слухачів, а це означає, що нова система числення вже була їм досить добре відома.

Отже, твори з мистецтва рахунку називалися Алгоритмами.
Поступово значення слова розширювалося. Вчені починали застосовувати його не тільки до суто обчислювальним, а й до інших математичних процедур. Наприклад, близько 1360 р французький філософ Микола Орем (Nicolaus Oresme, 1323 / 25-1382) написав математичний

трактат Algorismus proportionum («Обчислення пропорцій»), в якому вперше використав ступеня з дробовими показниками і фактично впритул підійшов до ідеї логарифмів.
В 1684 Готфрід Лейбніц в творі Nova Methodvs pro maximis et minimis, itemque tangentibus ... вперше використав слово «алгоритм» (Algorithmo) в ще більш широкому сенсі: як систематичний спосіб вирішення проблем диференціального обчислення.

У XVIII столітті в одному з німецьких математичних словників, виданому в Лейпцигу в 1747 р, термін algorithmus все ще пояснюється як поняття про чотирьох арифметичних операціях. Але таке значення не було єдиним, адже термінологія математичної науки в ті часи ще тільки формувалася. Зокрема, вираз algorithmus infinitesimalis застосовувалося до способів виконання дій з нескінченно малими величинами. Користувався словом алгоритм і Леонард Ейлер, одна з робіт якого так і називається - «Використання нового алгоритму для вирішення проблеми Пелля».

Однак треба було ще майже два сторіччя, щоб все старовинні значення слова вийшли з ужитку. Цей процес можна простежити на прикладі проникнення слова «алгоритм» в російську мову. Історики датують тисячі шістсот дев'яносто-один роком один зі списків давньоруського підручника арифметики, відомого як «Рахункова мудрість». Цей твір відоме в багатьох варіантах (найраніші з них майже на сто років старше) і сходить до ще більш стародавніх рукописів XVI в. За ним можна простежити, як знання арабських цифр і правил дій з ними поступово поширювалося на Русі.
Таким чином, слово «алгоритм» розумілося першими російськими математиками так само, як і в Західній Європі. Однак його не було ні в знаменитому словнику В. І. Даля, ні через сто років в «Тлумачному словнику російської мови» за редакцією Д. Н. Ушакова (1935 р). Зате слово «алгорифм» можна знайти і в популярному дореволюційному енциклопедичному словнику братів Гранат, і в першому виданні Великої радянської енциклопедії (Вікіпедія), виданому в 1926 р І там, і там воно трактується однаково: як правило, за яким виконується та чи інша з чотирьох арифметичних дій в десятковій системі числення. Однак до початку XX в. для математиків слово «алгоритм» вже означало будь-арифметичний або алгебраїчний процес, що виконується за строго визначеними правилами.
Алгоритми ставали предметом все більш пильної уваги вчених, і поступово це поняття посіло одне з центральних місць в сучасній математиці. Що ж стосується людей, від математики далеких, то до початку сорокових років це слово вони могли почути хіба що під час навчання в школі, в поєднанні «алгоритм Евкліда». Незважаючи на це, алгоритм все ще сприймався як термін суто спеціальний, що підтверджується відсутністю відповідних статей в менш об'ємних виданнях. Зокрема, його немає навіть в десятитомной Малої радянської енциклопедії (1957 р), не кажучи вже про однотомних енциклопедичних словниках. Але зате через десять років, в третьому виданні Великої радянської енциклопедії (1969 р) алгоритм вже характеризується як одна з основних категорій математики, «що не володіють формальним визначенням в термінах більш простих понять, і абстрагіруемие безпосередньо з досвіду». За сорок років алгоритм перетворився в одне з ключових понять математики, і визнанням цього стало включення слова в словники. Наприклад, воно присутнє в академічному "Словнику російської мови» саме як термін з області математики.
Одночасно з розвитком поняття алгоритму поступово відбувалася і його експансія з чистої математики в інші сфери. І початок їй поклало поява комп'ютерів, завдяки якому слово «алгоритм» увійшло в 1985 р в усі шкільні підручники інформатики і знайшло нове життя. Можна сказати, що його сьогоднішня популярність безпосередньо пов'язана зі ступенем поширення комп'ютерів.

Визначення алгоритму

Неформальне визначення

Кожен алгоритм передбачає існування початкових (вхідних) даних і в результаті роботи призводить до отримання певного результату. Робота кожного алгоритму відбувається шляхом виконання послідовності деяких елементарних дій. Ці дії називають кроками, а процес їх виконання називають алгоритмічним процесом. Таким чином проявляється властивість дискретності алгоритму.

Важливим властивістю алгоритмів є масовість, або можливість застосування до різних вхідних даних. Тобто, кожен алгоритм покликаний вирішувати клас однотипних завдань.

Необхідною умовою, якому задовольняє алгоритм, є детермінованість, або визначеність. Це означає, що виконання команд алгоритму відбувається за єдиним зразком і призводить до однакового результату для однакових вхідних даних.

Вхідні дані алгоритму можуть бути обмежені набором допустимих вхідних даних. Застосування алгоритму до неприпустимих вхідних даних може призводити до того, що алгоритм ніколи не зупиниться або потрапить в тупиковий стан (зависання), з якого не зможе вийти.

Формальне визначення

Різноманітні теоретичні проблеми математики та прискорення розвитку фізики і техніки поставили на порядок денний точне визначення поняття алгоритму.

Перші спроби уточнення поняття алгоритму і його дослідження здійснювали в першій половині XX століття Алан Тьюринг, Еміль Пост, Жак Ербран, Курт Гедель, Андрій Марков, Алонзо Черч. Було розроблено кілька визначень поняття алгоритму, але згодом було з'ясовано, що всі вони визначають одне і те ж поняття (див. Теза Черча - Тьюринга)

машина Тьюринга

Основна ідея, що лежить в основі машини Тьюринга, дуже проста. Машина Тьюринга - це абстрактна машина (автомат), що працює зі стрічкою окремих осередків, в яких записані символи. Машина також має головку для запису і читання символів з осередків, яка може рухатися уздовж стрічки. На кожному кроці машина зчитує символ з осередки, на яку вказує голівка, і, на основі ліченого символу і внутрішнього стану, робить наступний крок. При цьому, машина може змінити свій стан, записати інший символ в клітинку або пересунути голівку на одну клітинку вправо або вліво.

На основі дослідження цих машин була висунута теза Тьюринга (основна гіпотеза алгоритмів):
Деякий алгоритм для знаходження значень функції, заданої в деякому алфавіті, існує тоді і тільки тоді, коли функція обчислюється по Тьюрингу, тобто коли її можна обчислити на машині Тьюринга.

Ця теза є аксіомою, постулатом, і не може бути доведений математичними методами, оскільки алгоритм не є точним математичним поняттям.

ЛЕКЦІЯ 3.
АЛГОРИТМИ

Рекурсивні функції

З кожним алгоритмом можна зіставити функцію, яку він обчислює. Однак виникає питання, чи можна довільної функції зіставити машину Тьюринга, а якщо немає, то для яких функцій існує алгоритм? Дослідження цих питань привели до створення в 1930-х роках теорії рекурсивних функцій.

Клас обчислюваних функцій був записаний в образ, що нагадує побудова деякої аксіоматичної теорії на базі системи аксіом. Спочатку були обрані найпростіші функції, обчислення яких очевидно. Потім були сформульовані правила (оператори) побудови нових функцій на основі вже існуючих. Необхідний клас функцій складається з усіх функцій, які можна отримати з найпростіших застосуванням операторів. Подібно тези Тьюрінга в теорії обчислювальних функцій була висунута гіпотеза, яка називається тезу Черча:

Числова функція тоді і тільки тоді алгоритмічно обчислюється, коли вона частково рекурсивна.

Доказ того, що клас обчислюваних функцій збігається з обчислюваними по Тьюрингу, відбувається в два етапи: спочатку доводять обчислення найпростіших функцій на машині Тьюринга, а потім - обчислення функцій, отриманих в результаті застосування операторів.

Таким чином, неформально алгоритм можна визначити як чітку систему інструкцій, що визначають дискретний детермінований процес, який веде від початкових даних (на вході) до шуканого результату (на виході), якщо він існує, за кінцеве число кроків; якщо шуканого результату не існує, алгоритм або ніколи не завершує роботу, або заходить в глухий кут.

Нормальний алгоритм Маркова

Нормальний алгоритм Маркова - це система послідовних застосувань підстановок, які реалізують певні процедури отримання нових слів з базових, побудованих з символів деякого алфавіту. Як і машина Тьюринга, нормальні алгоритми не виконують самих обчислень: вони лише виконують перетворення слів шляхом заміни букв за заданими правилами.

Нормально обчислюється називають функцію, яку можна реалізувати нормальним алгоритмом. Тобто, алгоритмом, який кожне слово з безлічі допустимих даних функції перетворює в її вихідні значення.

Творець теорії нормальних алгоритмів А. А. Марков висунув гіпотезу, яка отримала назву принцип нормалізації Маркова:

Для знаходження значень функції, заданої в деякому алфавіті, тоді і тільки тоді існує певний алгоритм, коли функція нормально обчислюється.

Подібно тез Тьюринга і Черча, принцип нормалізації Маркова не може бути доведений математичними засобами.

Стохастичні алгоритми

Однак, наведене вище формальне визначення алгоритму в деяких випадках може бути занадто суворим. Іноді виникає потреба у використанні випадкових величин. Алгоритм, робота якого визначається не тільки вихідними даними, але і значеннями, отриманими з генератора випадкових чисел, називають стохастичним (або рандомізованих, від англ. Randomized algorithm). Формально, такі алгоритми можна називати алгоритмами, оскільки існує ймовірність (близька до нуля), що вони не зупиняться. Однак, стохастичні алгоритми часто бувають ефективніше детермінованих, а в окремих випадках - єдиним способом вирішити задачу.

На практиці замість генератора випадкових чисел використовують генератор псевдовипадкових чисел.

Однак слід відрізняти стохастичні алгоритми і методи, які дають з високою ймовірністю правильний результат. На відміну від методу, алгоритм дає коректні результати навіть після тривалої роботи.
Деякі дослідники допускають можливість того, що стохастичний алгоритм дасть з деякою заздалегідь відомою ймовірністю неправильний результат. Тоді стохастичні алгоритми можна розділити на два типи:
• алгоритми типу Лас-Вегас завжди дають коректний результат, але час їх роботи визначено.
• алгоритми типу Монте-Карло, на відміну від попередніх, можуть давати неправильні результати з відомою ймовірністю (їх часто називають методами Монте-Карло).
інші формалізації
Для деяких завдань названі вище формалізації можуть ускладнювати пошук рішень та здійснення досліджень. Для подолання перешкод були розроблені як модифікації «класичних» схем, так і створені нові моделі алгоритму. Зокрема, можна назвати:
• багатострічковій і недетермінірованного машини Тьюринга;
• реєстрова і РАМ машина - прототип сучасних комп'ютерів і віртуальних машин;
• кінцеві і клітинні автомати

Формальні властивості алгоритмів

Різні визначення алгоритму в явній або неявній формі містять наступний ряд загальних вимог:
• Дискретність - алгоритм повинен представляти процес вирішення завдання як послідовне виконання деяких простих кроків. При цьому для виконання кожного кроку алгоритму потрібно кінцевий відрізок часу, тобто перетворення вихідних даних в результат здійснюється в часі дискретно.
• детермінованість (визначеність). У кожен момент часу наступний крок роботи однозначно визначається станом системи. Таким чином, алгоритм видає один і той же результат (відповідь) для одних і тих самих вихідних даних. У сучасному трактуванні у різних реалізацій одного і того ж алгоритму повинен бути ізоморфний граф. З іншого боку, існують імовірнісні алгоритми, в яких наступний крок роботи залежить від поточного стану системи і генерується випадкового числа. Однак при включенні методу генерації випадкових чисел в список «вихідних даних», імовірнісний алгоритм стає підвидом звичайного.
• Зрозумілість - алгоритм для виконавця повинен включати тільки ті команди, які йому (виконавцю) доступні, які входять в його систему команд.
• завершаемості (кінцівку) - при коректно заданих вихідних даних алгоритм повинен завершувати роботу і видавати результат за кінцеве число кроків. З іншого боку, імовірнісний алгоритм може і ніколи не видати результат, але ймовірність цього дорівнює 0.
• Масовість (універсальність). Алгоритм повинен бути застосовний до різних наборів вихідних даних.
• Результативність - завершення алгоритму певними результатами.
• Алгоритм містить помилки, якщо призводить до отримання неправильних результатів або не дає результатів зовсім.
• Алгоритм не містить помилок, якщо він дає правильні результати для будь-яких допустимих вихідних даних.

Види алгоритмів

Особливу роль виконують прикладні алгоритми, призначені для вирішення певних прикладних задач. Алгоритм вважається правильним, якщо він відповідає вимогам завдання (наприклад, дає фізично правдоподібний результат). Алгоритм (програма) містить помилки, якщо для деяких вихідних даних він дає неправильні результати, збої, відмови або не дає ніяких результатів взагалі. Останню тезу використовується в олімпіадах з алгоритмічного програмування, щоб оцінити складені учасниками програми.
Важливу роль відіграють рекурсивні алгоритми (алгоритми, що викликають самі себе до тих пір, поки не буде досягнуто деяке умова повернення). Починаючи з кінця XX - початку XXI

століття активно розробляються паралельні алгоритми, призначені для обчислювальних машин, здатних виконувати кілька операцій одночасно.

Нумерація алгоритмів

Нумерація алгоритмів грає важливу роль в їх дослідженні та аналізі. Оскільки будь-який алгоритм можна задати у вигляді кінцевого слова (представити у вигляді кінцевої послідовності символів деякого алфавіту), а безліч всіх кінцевих слів в кінцевому алфавіті рахункова, то безліч всіх алгоритмів також рахункова. Це означає існування взаємно однозначного відображення між безліччю натуральних чисел і множиною алгоритмів, тобто можливість привласнити кожному алгоритму номер. Нумерація алгоритмів є одночасно і нумерацією всіх алгоритмічно обчислюваних функцій, причому будь-яка функція може мати нескінченну кількість номерів. Існування нумерації дозволяє працювати з алгоритмами так само, як з числами. Особливо корисна нумерація в дослідженні алгоритмів, що працюють з іншими алгоритмами.

Алгоритмічно нерозв'язна задача

Формалізація поняття алгоритму дозволила досліджувати існування завдань, для яких не існує алгоритмів пошуку рішень. Згодом було доведено неможливість алгоритмічного обчислення рішень ряду завдань, що робить неможливим їх рішення на будь-якому обчислювальному пристрої. Функцію f називають обчислюється (англ. Computable), якщо існує машина Тьюринга, яка обчислює значення f для всіх елементів множини визначення функції. Якщо такої машини не існує, функція f називають невичісляемой. Функція буде вважатися невичісляемой, навіть якщо існують машини Тьюринга, здатні обчислити значення для підмножини зі всієї безлічі вхідних даних.

Випадок, коли результатом обчислення функції f є логічне вираження «істина» або «брехня» (або безліч {0, 1}), називають завданням, яка може бути розв'язуваної або нездійсненним в залежності від обраховуються функції f.

Важливо точно вказувати допустиме безліч вхідних даних, оскільки завдання може бути розв'язуваної для одного безлічі і нездійсненним для іншого.

Однією з перших завдань, для якої була доведена не вирішується, є проблема зупинки. Формулюється вона наступним чином:

Маючи опис програми для машини Тьюринга, можна визначити, чи завершить роботу програма по закінченню часу або буде працювати нескінченно, отримавши деякі вхідні дані.

Доказ нерозв'язності проблеми зупинки важливо тим, що до неї можна звести інші завдання. Наприклад, проблему зупинки на порожній рядку (коли потрібно визначити для заданої машини Тьюринга, чи зупиниться вона, будучи запущеною на порожній рядку) можна звести до простого завдання зупинки, довівши тим самим її нерозв'язність.

Аналіз алгоритмів

Докази коректності

Разом з поширенням інформаційних технологій збільшився ризик програмних збоїв. Одним із способів уникнення помилок в алгоритмах і їх реалізаціях служать докази коректності систем математичними засобами.

Використання математичного апарату для аналізу алгоритмів і їх реалізацій називають формальними методами. Формальні методи передбачають застосування формальних специфікацій і, звичайно, набору інструментів для синтаксичного аналізу і доведення властивостей специфікацій. Абстрагування від деталей реалізації дозволяє встановити властивості системи незалежно від її реалізації. Крім того, точність і однозначність математичних тверджень дозволяє уникнути багатозначності і неточності природних мов.

За гіпотезою Річарда Мейса, «уникнути помилок краще усунення помилок». За гіпотезою Хоара, «доказ програм вирішує проблему коректності, документації і сумісності». Доведення коректності програм дозволяє виявляти їх властивості по відношенню до всього діапазону

вхідних даних. Для цього поняття коректності було розділене на два типи:
• Часткова коректність - програма дає правильний результат для тих випадків, коли вона завершується.
• Повна коректність - програма завершує роботу і видає правильний результат для всіх елементів з діапазону вхідних даних.
Під час докази коректності порівнюють текст програми зі специфікацією бажаного співвідношення вхідних-вихідних даних. Для доказів типу Хоара ця специфікація має вигляд тверджень, які називають передумовою і умовою поста. У сукупності з самою програмою, їх ще називають трійкою Хоара. Ці твердження записують $P\{Q\}R$, де P - це передумова, що має виконуватися перед запуском програми Q, а R - постусловієм, правильне після завершення роботи програми.

Час роботи

Поширеним критерієм оцінки алгоритмів є час роботи і порядок зростання тривалості роботи в залежності від обсягу вхідних даних.
Для кожного конкретного завдання складають деяке число, яке називають її розміром. Наприклад, розміром завдання обчислення добутку матриць може бути найбільший розмір матриць-множників, для задач на графах розміром може бути кількість ребер графа.
Час, який витрачає алгоритм як функція від розміру задачі n, називають тимчасовою складністю цього алгоритму T (n). Асимптотику поведінки цієї функції при збільшенні розміру завдання називають асимптотічность тимчасової складністю, а для її позначення використовують спеціальну нотацію.
Саме асимптотична складність визначає розмір завдань, які алгоритм здатний обробити. Наприклад, якщо алгоритм обробляє вхідні дані розміром n за час cn^2, де c - деяка константа, то кажуть, що тимчасова складність такого алгоритму O (n^2).
Часто, під час розробки алгоритму намагаються зменшити асимптотическую тимчасову складність для найгірших випадків. На практиці ж бувають випадки, коли достатнім є алгоритм, який «зазвичай» працює швидко.

Грубо кажучи, аналіз середньої асимптотической тимчасової складності можна розділити на два типи: аналітичний і статистичний. Аналітичний метод дає більш точні результати, але складний у використанні на практиці. Зате статистичний метод дозволяє швидше здійснювати аналіз складних завдань.

Складність	Коментар	Приклади
$O(1)$	Сталий час роботи не залежить від розміру завдання	Очікуваний час пошуку в в хеш-таблиці
$O(\log \log n)$	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів
$O(\log n)$	Логарифмічний зростання - подвоєння розміру задачі збільшує час роботи на постійну величину	Обчислення x^n; Двійковий пошук в масиві з n елементів
$O(n)$	Лінійний ріст - подвоєння розміру задачі подвоїть і необхідний час	Додавання / віднімання чисел з n цифр; Лінійний пошук в масиві з n елементів
$O(n \log n)$	Лінеарітмічний зростання - подвоєння розміру задачі збільшить необхідний час трохи більше ніж удвічі	Сортування злиттям або купою n елементів; нижня межа сортування зіставленням n елементів
$O(n^2)$	Квадратичний зростання - подвоєння розміру задачі збільшує необхідний час в чотири рази	Елементарні алгоритми сортування
$O(n^3)$	Кубічний зростання - подвоєння розміру задачі збільшує	Звичайне множення матриць

	необхідний час у вісім разів	
$O(c^n)$	Експоненціальне зростання - збільшення розміру завдання на 1 призводить до с-кратного збільшення необхідного часу; подвоєння розміру задачі збільшує необхідний час в квадрат	Деякі завдання комівояжера, алгоритми пошуку повним перебором

Наявність вихідних даних і деякого результату

Алгоритм - це точно певна інструкція, послідовно застосовуючи яку до вихідних даних, можна отримати рішення задачі. Для кожного алгоритму є деякий безліч об'єктів, допустимих в якості вихідних даних. Наприклад, в алгоритмі розподілу дійсних чисел ділене може бути будь-яким, а дільник не може бути дорівнює нулю.
Алгоритм служить, як правило, для вирішення не однієї конкретної задачі, а деякого класу задач. Так, алгоритм складання застосуємо до будь-якої парі натуральних чисел. У цьому виражається його властивість масовості, тобто можливості застосовувати багаторазово один і той же алгоритм для будь-якого завдання одного класу.
Для розробки алгоритмів і програм використовується алгоритмизация - процес систематичного складання алгоритмів для вирішення поставлених прикладних завдань. Алгоритмізація вважається обов'язковим етапом в процесі розробки програм та вирішенні завдань на ЕОМ. Саме для прикладних алгоритмів і програм принципово важливі детермінованість, результативність і масовість, а також правильність результатів вирішення поставлених завдань.
Алгоритм - це зрозуміле і точне розпорядження, виконавчо зробити послідовність дій, спрямованих на досягнення мети.

Подання алгоритмів

Форми запису алгоритму:
• словесна або вербальна (мовна, формульно-словесна);
• дракон-схема;
• псевдокод (формальні алгоритмічні мови);
• схематична:
• структурограмми (схеми Насс-Шнайдерман);
• графічна (блок-схеми, виконується з вимогами стандарту).
Зазвичай спочатку (на рівні ідеї) алгоритм описується словами, але в міру наближення до реалізації в неї з'являються все більш формальні обриси і формулювання мовою, зрозумілою виконавцю (наприклад, машинний код).

Ефективність алгоритмів

Хоча у визначенні алгоритму потрібно лише кінцівку числа кроків, необхідних для досягнення результату, на практиці виконання навіть хоча б мільярда кроків є занадто повільним. Також зазвичай є інші обмеження (на розмір програми, на допустимі дії). У зв'язку з цим вводять такі поняття як складність алгоритму (тимчасова, за розміром програми, обчислювальна та ін.).
Для кожного завдання може існувати безліч алгоритмів, що призводять до мети. Збільшення ефективності алгоритмів становить одну із завдань сучасної інформатики. У 50-х рр. XX століття з'явилася навіть окрема її область - швидкі алгоритми.
Яскравим прикладом є алгоритм Чуднівського для обчислення числа π.

ЛЕКЦІЯ 4.
ВСТУП ДО ОСНОВ ПРОГРАМУВАННЯ

ОСНОВНІ ПОНЯТТЯ ПРОГРАМУВАННЯ

Програмування – розробка програм за допомогою мов програмування.

Мова програмування – це формальна система знаків, що призначена для написання програм, зрозуміла для виконавця (комп'ютера).

Програма (program, routine) - впорядкована послідовність команд (інструкцій) комп'ютера для розв'язання задачі.

Програмне забезпечення (software) - сукупність програм обробки даних та необхідних для їх експлуатації документів. Програми призначені для машинної реалізації завдань (Задач). Терміни завдання і застосування (програма) мають дуже широке вживання в контексті інформатики і програмного забезпечення.

Завдання або задача (problem, task) - проблема, що підлягає вирішенню.
Застосування або програма (application) - програмна реалізація на комп'ютері рішення задачі.

Розробка програмного забезпечення (англ. software engineering, software development) - це рід діяльності (професія) і процес, спрямований на створення і підтримку працездатності, якості та надійності програмного забезпечення, використовуючи технології, методологію та практики з інформатики, управління проектами, математики, інженерії та інших сфер знань.

ПАРАДИГМИ ТА МОВИ ПРОГРАМУВАННЯ

Усе програмування прийнято поділяти на два основних види: *Декларативне* та *Імперативне*.

Декларативне програмування - термін з двома різними значеннями. Згідно першому визначенню, програма «декларативна», якщо вона описує щось, а не як його створити. Наприклад, веб-сторінки на HTML декларативні, оскільки вони описують що повинна містити сторінка, а не як відображати сторінку на екрані. Цей підхід відрізняється від мов імперативного програмування, що вимагають від програміста вказувати алгоритм для виконання.

Згідно другому визначенню, програма «декларативна», якщо вона написана на виключно функціональній, логічній або константній мові програмування.

Імперативне програмування — парадигма програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми. Подібно до того, як з допомогою наказового способу в мовознавстві перелічується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп'ютеру для виконання.

Парадигма програмування — це спосіб мислення розробника програми. Мова програмування може підтримувати або не підтримувати ту чи іншу парадигму. В першому випадку застосування парадигми стає зручним, тобто простим, безпечним і ефективним. Ми розглянемо три основних наказових парадигми — процедурне, об'єктне (модульне) і об'єктно-орієнтовне (ієрархічне) програмування.

ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

Процедурне програмування подає програму у вигляді набору алгоритмів, для оформлення яких можуть застосовуватися іменовані програмні блоки — процедури і функції. В

останньому випадку передбачається наявність механізмів передачі параметрів і поверненні результату.

Спочатку процедурне програмування користувалося довільними засобами керування, в тому числі, переходом за міткою — одним з найбільш вживаних операторів керування в Фортрані.

До мов процедурного програмування відносяться Fortran, Cobol, Pascal, Basic, та інші.

В 1968 році голландський вчений Е. Дейкстра вперше звернув увагу на проблеми, що виникають у програмах з неконтрольованими переходами, в 1970 році проголосив новий напрямок, який він назвав структур(ова)ним програмуванням.

Структурне програмування — це варіант процедурного, що вживає три типи структур керування: послідовне виконання дій, розгалуження і цикл. Не дивно, що Фортран не підтримував цю парадигму — в наборі його засобів не було циклів за умовами. Починаючи з Алголу, а особливо в Паскалі, цикли стають основним засобом організації обчислень в програмі.

Автор Паскалю, професор Н. Вірт, відібрав до створюваної ним мови програмування лише прості в поясненні і легкі в реалізації конструкції. Завдяки сильній типізації програми в Паскалі відзначаються високою надійністю, вони мобільні завдяки закладеній в них концепції Паскаль-машини, їх легко читати і розуміти завдяки дисципліні програмування, продиктованої вжитою парадигмою.

Але разом з цим застосування Паскалю гальмувалося саме складністю виходу за межі віртуальної машини, потребою ефективного використання наявної апаратури. Головним критерієм, вжитим Б.Кернiганом і Д.Річі до створеної ними мови С, стала саме гнучкість використання особливостей конкретної апаратури і ефективність виконання програм.

ОБ'ЄКТНЕ (МОДУЛЬНЕ) ПРОГРАМУВАННЯ

Процедурна парадигма віддала належне алгоритмічній компоненті програмування. Але з ростом обсягу програм і складності даних з'явилася нова проблема структурної організації даних, найбільш ємко висловлена Віртовською формулою “алгоритми + структури даних = програми”.

Поняття модуля як абстракції даних було вперше запропоноване Парнасом у 1972 році, правда на той час уже існувала мова програмування Симула 67, в якій використовувалася парадигма об'єктів. У найбільш повному виді поняття абстракції даних було реалізоване в мові програмування Модула-2.

Головна ідея полягає в забезпеченні доступу до даних, не залежному від їх конкретного представлення. Самі дані і програми їх обробки вбудовуються (інкапсулюються) в окремій одиниці програми.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Об'єктно-орієнтована парадигма наділила класи ієрархією. Об'єктно-орієнтоване програмування за метафорою Б.Страуструпа, автора С++ — однієї з найпопулярніших мов об'єктно-орієнтованого програмування, — це високоінтелектуальний синонім доброго програмування. Дійсно, нові парадигми програмування з'являються не так часто, не частіше однієї десятиліття. Той факт, що об'єктно-орієнтована парадигма успішно використовується протягом 20 років, сам по собі служить вагомим підтвердженням її життєздатності.

Алгоритми, реалізовані в процедурному програмуванні, надто конкретні. Будь-яка модифікація — це вже новий алгоритм і таким чином кількість процедур і функцій, що знаходяться у використанні, надмірно зростає. Модульне програмування групує алгоритми в модулі, одночасно інкапсулюючи структури даних. Тепер залишається зробити наступний крок — побудувати ієрархію модулів або класів.

Таких ієрархій може бути дві. Перша з них — бути частиною чогось. Наприклад, грань є частиною многогранника, ребро — частиною грані, вершина — частиною ребра. Інша ієрархія — бути узагальненням або конкретизацією. Наприклад, овал і многокутник служать конкретизацією плоскої фігури, коло — конкретизацією овалу, чотирикутник — конкретизацією

многокутника, подальшими конкретизаціями чотирикутника можуть служити паралелограм, прямокутник, ромб, квадрат. Той факт, що квадрат, ромб, прямокутник є повноцінними паралелограмами дозволяє їм користуватися усіма програмними засобами, створеними для паралелограма, паралелограм в свою чергу є повноцінним чотирикутником і так далі. Цей принцип, відомий під назвою reusable — знову вживаний — став одним з найважливіших досягнень об'єктно-орієнтованої парадигми. Знову вживаючи вже існуюче програмне забезпечення в більш конкретизованих умовах, ми дописуємо лише ту його частину, яка стосується особливостей наявної конкретизації. Цей принцип дістав назву programming by difference або дописування програм.

І, нарешті, об'єктно-орієнтована парадигма доводить до логічної завершеності принцип моделювання реального світу, а точніше тієї його частини, абстракцією якої служить програма. При цьому підході програма складається з об'єктів, що відповідають реальним поняттям або предметам. Виконання програми зводиться до взаємодії об'єктів, яке служить абстракцією реальної взаємодії їх прототипів. Все це разом забезпечило об'єктно-орієнтованому підходу беззаперечне лідерство в галузі розробки програм.

Сьогодні в сімействі мов об'єктно-орієнтованого програмування три найбільш відомих представника: C++, Java і C#. C++ і сьогодні залишається визнаним лідерів в розробці великих і складних програмних систем. Java і C # виросли з C++. Вони мають свою сферу застосування в розподіленому програмуванні і будуть вивчатися нами пізніше.

СТВОРЕННЯ ООП І C++

До кінця 1970-х розмір проектів став наближатися до критичного, при перевищенні якого методика структурного програмування і мова С "опускали руки". Тому стали з'являтися нові підходи до програмування, що дозволяють < вирішити цю проблему. Один з них отримав назву об'єктно-орієнтованого програмування (ООП). Використовуючи ООП, програміст міг справлятися з програмами набагато більшого розміру, ніж раніше. Але проблема полягала у тому, що С, найпопулярніша на той час мова, не підтримувала ООП. Бажання працювати з об'єктно-орієнтованою версією мови С врешті-решт і привело до створення C++.

Мова C++ була розроблена Бьорном Страуструпом (Bjarne Stroustrup) в компанії Bell Laboratories (Нью-Джерсі), роком створення вважається 1979-й. Спочатку творець нової мови назвав її "С з класами", але в 1983 році це ім'я було змінене на C++. C++ повністю включає елементи мови С. Таким чином, С можна вважати фундаментом, на якому побудований C++. Більшість доповнень, які Страуструп вніс до С, були призначені для підтримки об'єктно-орієнтованого програмування. По суті, C++ - це об'єктно-орієнтована версія мови С. Створюючи C++ на основі С Страуструп забезпечив плавний перехід багатьох програмістів на ООП. Замість необхідності вивчати абсолютно нову мову, С-програмістові досить було освоїти лише нові засоби, що дозволяють використовувати переваги об'єктно-орієнтованої методики.

Впродовж 1980-х років C++ інтенсивно розвивалася і на початок 1990-х вже була готовп для широкого використання. Зростання її популярності носило вибухоподібний характер, і до кінця цього десятиліття вона стала найбільш використовуваною мовою програмування. В наші дні мова C++ як і раніше має безперечну перевагу при розробці високопродуктивних програм системного рівня.

INTERNET І ПОЯВА МОВИ JAVA

Наступним рівнем на сходах прогресу мов програмування стала мова Java, яка спочатку називалася Oak (у перекладі з англ. "дуб"). Робота над її створенням почалася в 1991 році в компанії Sun Microsystems. Основною рушійною силою розробки Java був Джеймс Гослінг (James Gosling).

Java - це структурна об'єктно-орієнтована мова програмування, синтаксис і принципи якої "родом" з C++. Своїми новаторськими аспектами Java зобов'язана не стільки прогресу в мистецтві програмування (хоча і це мало місце), скільки змінам в комп'ютерному середовищі. Ще до настання ери Internet більшість програм писалися, компілювалися і призначалися для

виконання з використанням певного процесора і під управлінням конкретної операційної системи. Не дивлячись на те що програмісти завжди прагнули робити свої програми так, щоб їх можна було застосовувати неодноразово, можливість легко переносити програму з одного середовища в іншу не була ще досягнута, до того ж проблема переносимості постійно вікладалися, вирішувалися ж більш насущні проблеми. Проте з появою всесвітньої мережі Internet, в якій виявилися зв'язаними різні типи процесорів і операційних систем, стара проблема портативності заявила про себе вже в повний голос. Для її вирішення знадобилася нова мова програмування, і нею стала Java

Цікаво відзначити, що, хоча єдиним найбільш важливим аспектом Java (і причиною швидкого визнання) є можливість створювати на ній кросплатформений (сумісний з декількома операційними середовищами) переносимий програмний код, вихідним імпульсом для виникнення Java стала не мережа Internet, а наполеглива потреба у незалежній від платформи мові, яку можна було б використовувати в процесі створення програмного забезпечення для вбудованих контролерів. У 1993 році стало очевидним, що проблеми міжплатформеної переносимості, що чітко виявилися при створенні коду в вбудованих контролерів, також виявилися актуальними при спробі написати код для Internet. Адже Internet - це величезне комп'ютерне середовище, в якому "мешкає" безліч комп'ютерів різних типів. І виявилось, що одні і ті ж методи вирішення проблеми переносимості в малих масштабах можна успішно застосувати і до набагато більших, тобто в Internet У Java переносимість досягається за допомогою перетворення вихідного коду програми в проміжний код, що іменується байт-кодом), тобто машинно-незалежний код, що генерується Java-компілятором. Байт-код виконується віртуальною машиною Java (Java Virtual Machine - JVM) - спеціальною операційною системою. Отже, Java-программа могла б працювати в будь-якому середовищі, де доступна JVM. А оскільки JVM відносно проста для реалізації, вона швидко стала доступною для великої кількості середовищ.

Використання Java-програмами байт-коду радикально відрізняло їх від C- і C++-програм. Якщо C/C++-программу потрібно виконати в іншій системі, її необхідно перекомпілювати в машинний код, відповідний цьому середовищу. Отже, аби створити C/C++-програму, призначену для виконання в різних середовищах, необхідно мати декілька різних виконуваних (машинних) версій цієї програми. Це було непрактично і дорого. І навпаки, використання для виконання Java-программ проміжної мови було елегантним і рентабельним рішенням. Саме це рішення було адаптоване для мови C#.

СТВОРЕННЯ C#

Розробники Java успішно вирішили багато проблем, пов'язаних з переносимістю в середовищі Internet, але далеко не всі. Одна з них - міжмовна можливість взаємодії (cross-language interoperability) програмних і апаратних виробів різних постачальників, або багатомовне програмування (mixed-language programming). В разі вирішення цієї проблеми програми, написані на різних мовах, могли б успішно працювати одна з іншою. Така взаємодія необхідна для створення великих систем з розподіленим програмним забезпеченням (ПЗ), а також для програмування компонентів ПЗ, оскільки найціннішим є компонент, який можна використовувати у широкому діапазоні комп'ютерних мов і операційних середовищ.

Крім того, в Java не досягнута повна інтеграція з платформою Windows. Хоча Java-програми можуть виконуватися в середовищі Windows (за умови встанолення віртуальної машини Java), Java Windows не є міцно зв'язаними середовищами. А оскільки Windows - це найбільш широко використовувана операційна система в світі, відсутність прямої підтримки Windows - серйозний недолік Java.

Аби задовольнити ці потреби, Microsoft розробила мову C#, C# була створена в кінці 1990-х років і стала частиною спільної .NET-стратегії Microsoft. Вперше вона побачила світло як альфа-версія в середині 2000 року. Головним архітектором C# був Андерс Хейлсберг (Anders Hejlsberg) - один з провідних фахівців в області мов програмування, що отримав визнання у всьому світі. Досить сказати, що в 1980-х він був автором успішного продукту Turbo Pascal, витончена реалізація якого встановила стандарт для всіх майбутніх компіляторів.

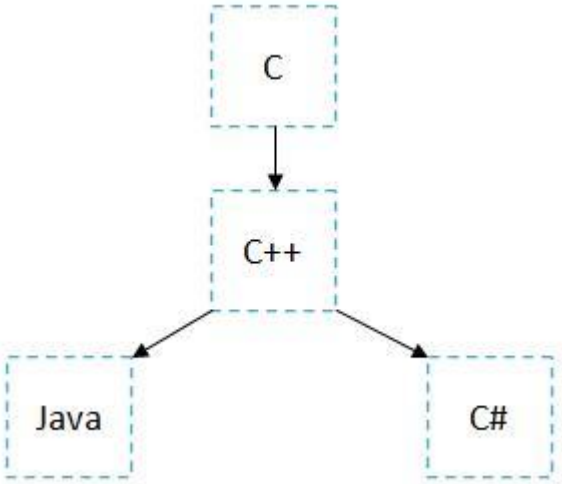


Рис. Розвиток мов програмування.

C# безпосередньо пов'язаний із C, C++ і Java. І це не випадково. Ці три мови - найпопулярніші і найулюбленіші мови програмування в світі. Більш того, майже всі професійні програмісти сьогодні знають C і C++, і більшість знає Java. Оскільки C# побудований на міцному, зрозумілому фундаменті, то перехід від цих "фундаментальних" мов до "надбудови" відбувається без особливих зусиль з боку програмістів. Оскільки Андерс Хейлсберг не збирався винаходити нову мову, він зосередився на введенні удосконалень.

Прамовою C# є мова C. Від C мова C# успадкувала синтаксис, багато ключових слів і оператори. Крім того, C# побудований на покращеній об'єктній моделі, визначеній в C++.

C# і Java зв'язані між собою дещо складніше. Як згадувалося вищим, Java також є нащадком і C++. У неї теж загальний з нею синтаксис і схожа об'єктна модель. Подібно Java C# призначений для створення переносимого коду. Проте C# - не нащадок Java. Швидше C# і Java можна вважати двоюрідними братами, що мають загальних предків, але що отримали від батьків різні набори "генів".

.NET Framework у основних поняттях.

Платформа - у контексті інформаційних технологій - середовище, що забезпечує виконання програмного коду. Платформа визначається характеристиками процесорів, особливостями операційних систем.

Framework - це інфраструктура середовища виконання програм, щось, що визначає особливості розробки і виконання програмного коду на даній платформі. Передбачає засоби організації взаємодії з операційною системою і прикладними програмами, методи доступу до баз даних, засоби підтримки розподілених (мережевих) додатків, мови програмування, безліч базових класів, уніфіковані інтерфейси користувача, парадигми програмування.

Microsoft .NET - платформа.

.NET Framework - інфраструктура платформи Microsoft .NET. Включає наступні основні компоненти: Common Language Runtime (CLR) і .NET Framework Class Library (.NET FCL).

CLS (Common Language Specification) - загальна специфікація мов програмування. Це набір конструкцій і обмежень, які є інструкцією для розробників бібліотек і компіляторів в середовищі .NET Framework. Бібліотеки, побудовані відповідно до CLS, можуть бути використані у будь-якій мові програмування, що підтримує CLS.

Мови, відповідні CLS (до їх числа відносяться мови Visual C# 2.0, Visual Basic, Visual C++), можуть інтегруватися один з одним. CLS - це основа міжмовної взаємодії в рамках платформи Microsoft .NET.

CLR (Common Language Runtime) - Середовище Часу Виконання або Віртуальна Машина. Забезпечує виконання збірки. Основний компонент .NET Framework. Під Віртуальною

Машиною розуміють абстракцію інкапсульованої (відособленої) керованої операційної системи високого рівня, яка забезпечує виконання (керованого) програмного коду.

Керований код - програмний код, який під час виконання здатний використовувати служби, що надаються CLR. Відповідно, некерований код подібною здатністю не володіє.

Тобто, CLR - це набір служб, необхідних для виконання керованого коду. Сама CLR складається з двох головних компонентів: ядра (mscoree.dll) і бібліотеки базових класів (mscorlib.dll). Наявність цих файлів на диску - ознака того, що на комп'ютері, принаймні, була зроблена спроба встановлення платформи .NET.

FCL (.NET Framework Class Library) - відповідна CLS-специфікаціїї об'єктно-орієнтована бібліотека класів, інтерфейсів і системи типів (типів-значень), які включаються до складу платформи Microsoft .NET. Ця бібліотека забезпечує доступ до функціональних можливостей системи і призначена служити основою при розробці .NET-додатків, компонент, елементів управління.

.NET бібліотека класів є другим компонентом CLR. .NET FCL можуть використовувати усі .NET-додатки, незалежно від призначення архітектури використовуваного при розробці мови програмування, і зокрема:

вбудовані (елементарні) типи, представлені у вигляді класів (на платформі .NET все побудовано на структурах або класах);

класи для розробки графічного користувацького інтерфейсу (Windows Forms);

класи для розробки web-додатків і web-служб на основі технології ASP.NET (Web Forms);

класи для розробки XML і Internet-протоколів (FTP, HTTP, SMTP, SOAP);

класи для розробки додатків, що працюють з базами даних (ADO .NET) і багато що інших.

.NET-застосування - програма, розроблена для виконання на платформі Microsoft .NET. Реалізується на мовах програмування, відповідних CLS.

MSIL (Microsoft Intermediate Language) - проміжна мова платформи Microsoft .NET. Вихідні тексти програм для .NET-додатків пишуться на мовах програмування, відповідних специфікації CLS. Для таких мов може бути побудований перетворювач в MSIL. Таким чином, програми на цих мовах можуть транслюватися в проміжний код на MSIL. Завдяки відповідності CLS, в результаті трансляції програмного коду, написаного на різних мовах, виходить сумісний IL-код.

У середовищі CLR допускається спільна робота і взаємодія компонентів програмного забезпечення, реалізованих на різних мовах програмування. CLR бере на себе вирішення багатьох проблем, які традиційно знаходилися в зоні особливої уваги розробників програмного забезпечення.

До функцій, виконуваних CLR, відносяться:

1. Перевірка і динамічна (JIT) компіляція MSIL-коду у команди процесора.
2. Управління пам'яттю, процесами і потоками.
3. Організація взаємодії процесів.
4. Вирішення проблем безпеки (в рамках безпеки, що існує в системі політики).

ЛЕКЦІЯ 5.
Основи роботи з Visual Studio та платформою .Net

Коротка характеристика та історія розвитку середовища розробки visual studio.

Microsoft Visual Studio - лінійка продуктів компанії Майкрософт, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів.

Середовище розробки Visual Studio представляє собою повний набір інструментів для створення як настільних додатків, так і корпоративних веб-додатків для спільної роботи груп. Використовуючи ефективні інструменти розробки Visual Studio, засновані на використанні компонентів, та інші технології, можна не тільки створювати ефективно працюючі настільні додатки, але і спрощувати спільне проектування, розробку і розгортання корпоративних рішень.

Visual Studio включає один або декілька компонентів з наступних:
Visual Basic .NET, а до його появи - Visual Basic
Visual C++
Visual C#
Visual F # (включений у Visual Studio 2010)

Багато варіантів поставки також включають:
Microsoft SQL Server або Microsoft SQL Server Express

Microsoft Visual Studio Express - лінійка безкоштовних інтегрованих середовищ розробки, полегшена версія Microsoft Visual Studio, розробленої компанією Microsoft . Відповідно до твердження Microsoft, «Express»-редакції пропонують налагоджену, просте у навчанні та використанні середовище розробки користувачам, які не є професійними розробниками ПЗ, - любителям та студентам.

Як і більшість сучасних програм, середовище розробки Visual Studio містить меню та набір інструментальних панелей. У лівій частині середовища розробки присутній елемент управління із позначкою вікна Toolbox.

У правій частині екрана знаходиться вікно Solution Explorer. У ньому можна побачити, з яких проектів складається рішення і які файли входять до складу цих проектів.

Нижче вікна Solution Explorer розташоване вікно властивостей (Properties). Це вікно містить список атрибутів об'єкта, виділеного в даний момент.

Давайте з'ясуємо, навіщо потрібні ці та інші вікна середовища розробки. Для кращого розуміння я буду використовувати приклад вікна програми із проектом на Windows-формах. (рис.)

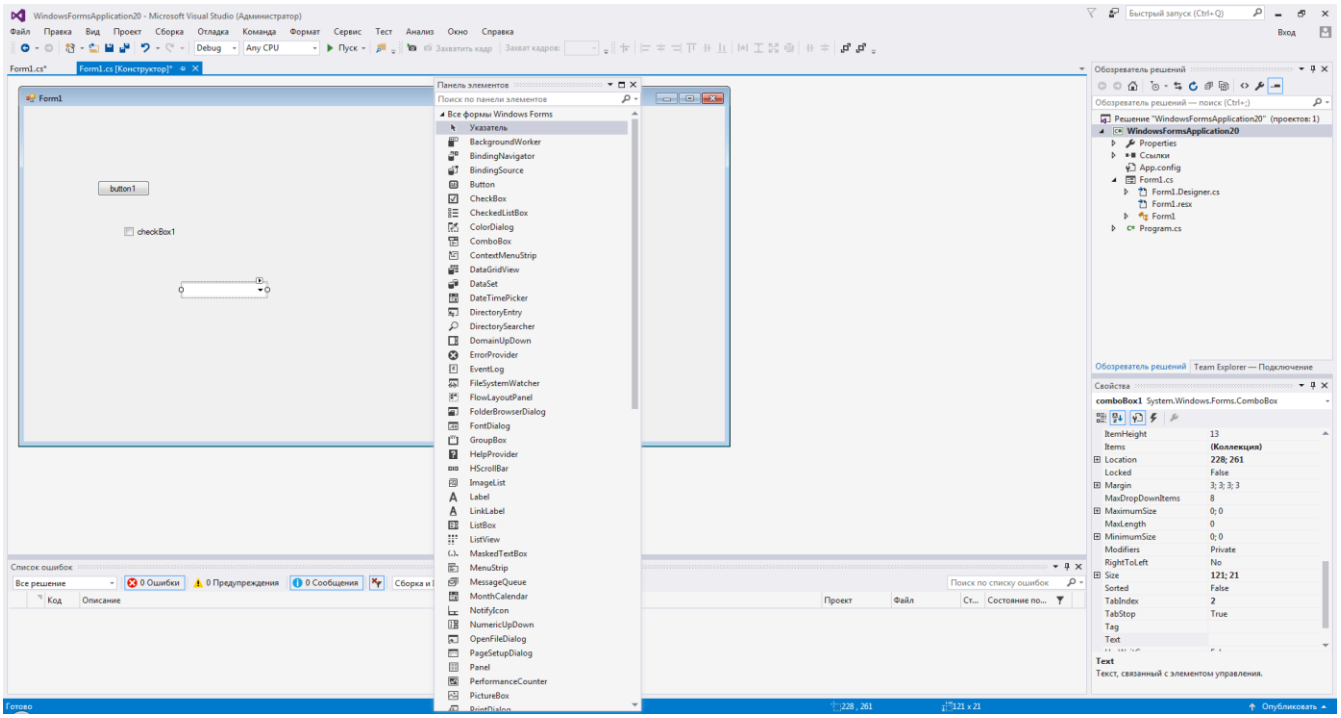


Рис. Робочий проект у Visual Studio

TOOLBOX - ПАНЕЛЬ КОМПОНЕНТІВ ДЛЯ РОЗРОБКИ.

У вікні Toolbox (його можна відобразити на екрані за допомогою команди меню View/Toolbox) знаходиться список елементів управління, які можна використовувати на формах програми. Тобто, який набір компонентів доступний в даний момент, залежить від типу проекту, що розробляється. Наприклад, якщо в даний момент розробляється програма типу Windows Forms, в цьому вікні будуть присутні елементи керування, які можна використовувати в Windows-застосуваннях; якщо ж розробляється Web-форма, в цьому вікні будуть знаходитися інструменти для роботи з елементами управління Web Controls, і т.д.

При необхідності можна змінити відображений у вікні Toolbox набір елементів управління, додавши інші компоненти. NET або елементи ActiveX (у тому числі створені незалежними виробниками). Для цієї мети можна використовувати команду меню Tools/Choose Toolbox Items

і за допомогою діалогової панелі Customize Toolbox вибрати елементи керування ActiveX або елементи керування. NET, які ми хочемо відобразити у вікні Toolbox.

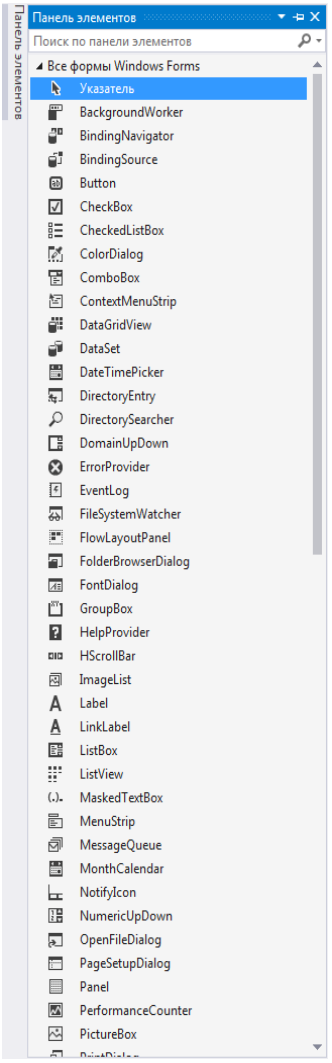


Рис. Вікно ToolBox

ВІКНО SOLUTION EXPLORER - ОГЛЯД РІШЕННЯ.

Рішення - це набір проектів, з яких складається програма. Вікно Solution Explorer (яке можна відобразити на екрані за допомогою команди меню View / Solution Explorer) дозволяє переглядати склад проектів, що входять до рішення, у вигляді ієрархічної структури, а також зв'язки між проектами та їх компонентами (рис.). Компонентами проектів можуть бути форми, класи, модулі, а також інші файли, які потрібні для створення програми. Якщо потрібно відредагувати компонент проекту, слід двічі клацнути по його імені у вікні Solution Explorer.

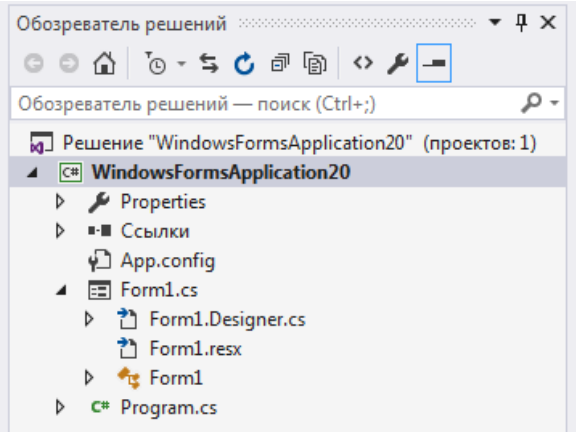


Рис. Вікно Solution Explorer.

За допомогою кнопок, розташованих у верхній частині вікна Solution Explorer, можна вказати, що саме повинно відображатися у середовищі розробки:

View Code - код, пов'язаний з файлом, виділеним у вікні Solution Explorer;
View Designer - дизайнер (візуальний редактор) файлу, виділеного у вікні Solution Explorer;
Refresh - оновити вміст вікна Solution Explorer;
Show All Files - всі файли, включаючи код, пов'язаний з формами;
Properties - властивості обраного файлу.

ВІКНО CLASS VIEW

Вікно Class View (доступно за допомогою команди меню View / Class View) дозволяє переглянути список властивостей і методів створених у застосуванні класів. Вибравши властивість або метод, можна клацнути на його імені правою кнопкою миші і вибрати одне з можливих дій з даними властивістю або методом. При подвійному натисканні кнопки миші по імені класу відбудеться його завантаження в редактор коду.

ВІКНО PROPERTIES

Вікно Properties (команда меню View / Properties Window) призначено для зміни властивостей елементів управління і інших класів створюваного програми. Властивості можна відсортувати за алфавітом або за категоріями (для цієї мети у верхній частині цього вікна є відповідні кнопки). Редагування властивостей може здійснюватися шляхом введення значення, вибору його з випадаючого списку або за допомогою установки його значення в окремій діалогової панелі - це залежить від типу конкретного властивості. Таким же чином можна змінювати властивості проекту, програми і т.п.

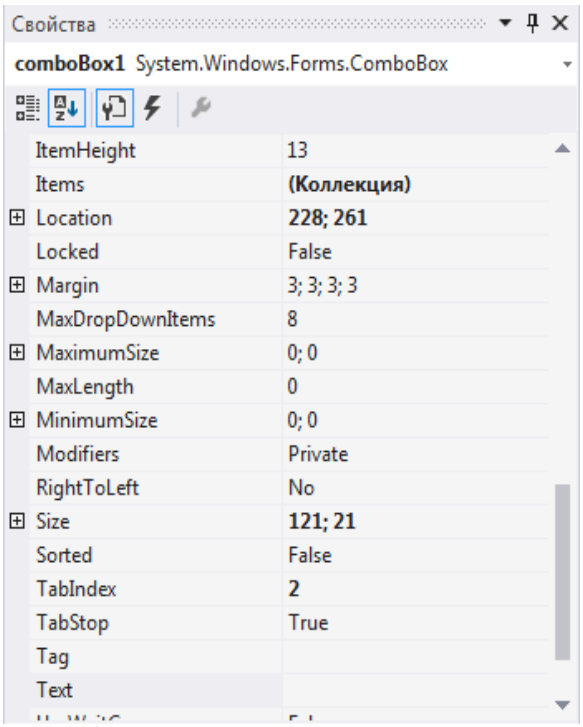


Рис. Вікно Properties.

ВІКНО ERROR LIST

Вікно Error List (команда меню View / Error List) призначено перегляду помилок, що були згенеровані середовищем розробки, а також застережень або корисних повідомлень. У цьому вікні можна дізантися про характер та причину помилки, а також у окремих випадках і як її вирішити.

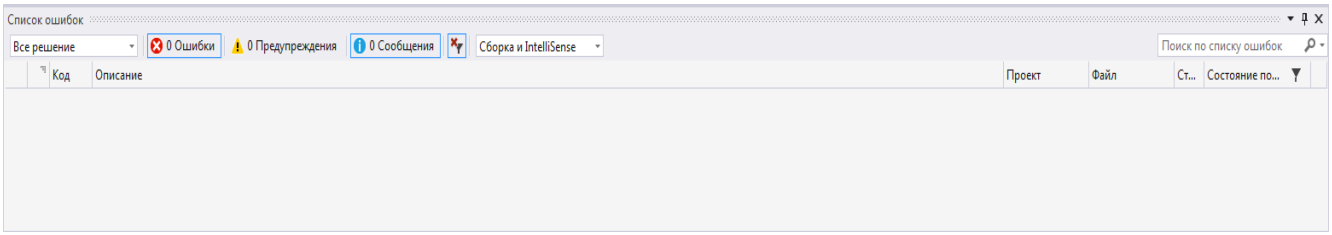


Рис. Вікно Error List.

ЛЕКЦІЯ 6.
Основи роботи з Visual Studio та платформою .Net

РЕЖИМИ РОБОТИ СЕРЕДОВИЩА РОЗРОБКИ

Середовище розробки Visual Studio містить два типи вікон - вікна інструментів і вікна документів. Вікна інструментів (частина з яких була описана вище) доступні за допомогою команд меню View та деяких інших, і їх доступність залежить від типу програми, але від того, які модулі розширення (додаткові утиліти та інструменти, в тому числі розроблені сторонніми розробниками) додані до середовища розробки. У вікнах ж документів можна редагувати компоненти проектів. Над вікнами інструментів та вікнами інструментів можна проводити 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 його використання підтримується операційною системою.

Деякі вікна інструментів, наприклад вікно Web Browser, можна створювати у кількох примірниках (це можна зробити, вибравши пункт меню Windows / New Window). Можна також змусити вікна інструментів автоматично зникати, якщо вони в даний момент не є активними, - в цьому випадку на екрані відображаються назва і піктограма вікна, над якою можна помістити курсор миші, якщо вікно потрібно відобразити цілком. Якщо необхідно запобігти зникненню вікна з екрану слід клацнути мишею по зображенню канцелярської кнопки на заголовку вікна.

Створення першого проекту у VS, структура програми на C#.

Для того, щоб створити новий проект потрібно виконати наступні дії:

1. File / New Project.
2. Обираємо Console Application (рис.).
3. Вводимо назву проекту / ОК.

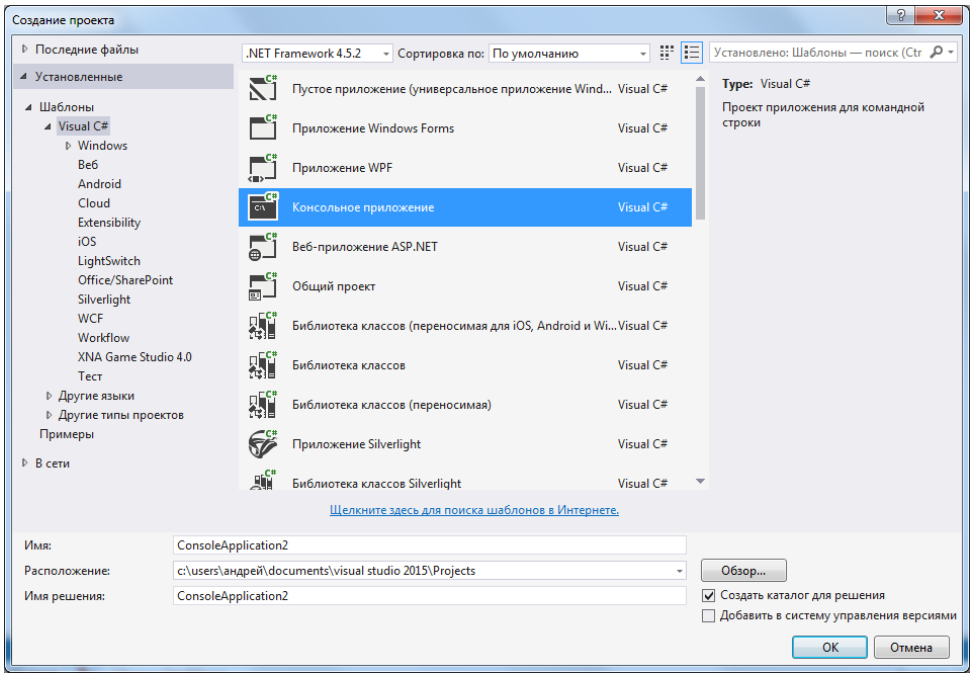


Рис. Вибір типу проекту.

У VS можна працювати з наступними видами проектів (рис.):
Windows Forms Application - застосування на формах, класина програма для виконання під ОС Windows.
WPF Application - застосування на формах на основі нової технології від компанії Майкрософт Windows Presentation Foundation.
Console Application - консольна програма.
Class Library - бібліотека класів .NET.
WPF Browser Application - WPF-застосування, що виконується у вікні веб-браузера.
Empty Project - пустий проект та інші.

Код першої програми у Visual Studio.

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

namespace ConsoleApplication2
{
   class Program
   {
      static void Main(string[] args)        {          }
   }
}
```

Для виведення на екран фрази "Hello World from C#" допишемо програму наступним чином:

Hello World from C#.

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

namespace ConsoleApplication2
{
   class Program
   {
      static void Main(string[] args)
      {
                  Console.WriteLine("Hello World from C#");

      }
   }
}
```

Далі потрібно відкомпілювати нашу програмиу (Ctrl+F5) і отримаємо вікно як на рис.

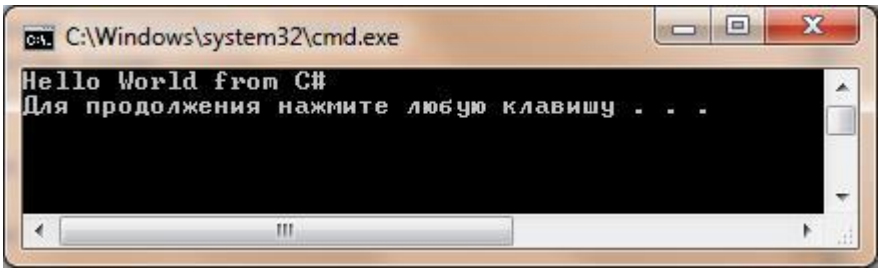


Рис. Результат виконання першої програми.

Для впорядкування і оформлення коду у мові програмування C# використовуються класи. Весь виконуваний код програми повинен міститися у класі.

ДИРЕКТИВИ USING І ПРОСТОРИ ІМЕН

При створення консольного застосування у перших лініях у редакторі коду містяться директиви using з перерахунком просторів імен .NET Framework.

Простір імен дозволяє, в деякому розумінні, згрупувати разом класи і структури, що обмежує їх зону дії і дозволяє уникнути конфлікту імен з іншими класами і структурами. При створенні програми в Visual C# простір імен створюється автоматично. При створенні нового застосування часто використовувані простори імен .NET Framework включені в список за замовчуванням. При використанні класів з інших просторів імен у бібліотеці класів необхідно додати директиву using для простору імен до вихідного файлу.

МЕТОД MAIN

У програмі на C# має бути присутнім метод Main, в якому починається і закінчується управління, але це не завжди!!! У методі Main створюються об'єкти і виконуються інші методи. Метод Main є статичним методом, розташованим усередині класу або структури. У попередньому прикладі "Hello World!" він розташований в класі з ім'ям Program. Метод Main можна оголосити одним із наступних способів:

повертає значення void (тобто, не повертає нічого).

```
1:  static void Main()
2:  {
3:      //...
4:  }
```

може повертати значення типу int.

```
5:  static int Main()
2:  {
3:      //...
4:      return 0;
5:  }
```

може приймати аргументи.

```
1:  static void Main(string[] args)
2:  {
```

```
3:      //...
4: }
```

або

```
1: static int Main(string[] args)
2: {
3:      //...
4:      return 0;
5: }
```

ПРОСТОРИ ІМЕН (NAMESPACES) В C#

Простори імен (namespaces) надають програмістам можливість логічного взаємозв'язку класів і інших типів. Саме поняття namespaces швидше є логічним, ніж фізичним (наприклад як файл або компонент). Кожного разу коли ви оголошуєте клас в C# - у вас є можливість додати його в простір імен. Коли стане необхідно розширювати функціональну частину програми - ви з легкістю зможете додати декілька логічно об’єднаних класів в один простір імен.

Для прикладу додамо структуру Book в простір імен BookStore:

```
1: namespace BookStore
2: {
3:      public struct Book
4:      {
5:          // Код структури ...
6:      }
7: }
```

Якщо надалі потрібно буде ще додавати класи і структури, які логічно пов'язані з продажем книг, - правильно буде додавати їх в простір імен BookStore. Кожне ім'я у просторі імен складається з назв тих просторів імен, в яке воно входить і починається з самого зовнішнього імені. Для прикладу: System.Windows.Forms. Такий принцип іменування може дати нам дуже довгі конструкції, які незручно використовувати в коді (System.Windows.Forms.Form) для цього в C# існує директива using, яка пишеться на самому початку коду має вигляд: using System.Windows.Forms;. Якщо ми підключили за допомогою директиви using простір імен System.Windows.Forms - то у коді вже можна безпосередньо звертатися до Form (без повного запису).

Важливою особливістю просторів імен в C# є також і те, що вони не залежать від збірок. Можна створювати різні простори в межах однієї збірки так само, як і створювати декілька збірок в межах одного простору імен. Ще одним вживанням ключового слова using в C# є призначення псевдонімів класам і просторам імен. Якщо у вас складний проект і простори імен мають дуже велику вкладеність - у такому разі простору імен призначається псевдонім, який в загальному вигляді виглядає так: using alias = NamespaceName;

ЛЕКЦІЯ 7.
ЗБІРКИ У .NET FRAMEWORK

Збірка (assembly) - це логічна одиниця, що містить скомпільований код для .NET Framework.

Збірка (assembly) - це повністю самодостатній і, швидше, логічний, ніж фізичний елемент. Це означає, що він може бути збережений в більш ніж одному файлі (хоча динамічні складки зберігаються в пам'яті, а зовсім не у файлах). Якщо збірка зберігається в більш ніж одному файлі, то мають бути один головний файл, що містить точку входу і що описує інші файли.

Слід зазначити, що одна і та ж структура збірки використовується як для виконуваного коду (*.exe), так і для коду бібліотек (*.dll). Єдина реальна відмінність виконуваної збірки полягає в тому, що вона містить головну точку входу програм, тоді як бібліотечна збірка - ні.

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

Збірки бувають двох видів: *розділені* і *приватні*.

Приватні збірки. Це простий тип збірок. Зазвичай вони поставляються з певним програмним забезпеченням і призначені для використання лише в його складі. Звичайний сценарій отримання приватної збірки - це коли вам поставляється застосування у вигляді виконуваної програми і бібліотек, код яких може бути використаний лише цим застосуванням.

Розділені збірки. Далі розглянемо розділені збірки, які більш універсальні і цікаві з точки зору реалізації. Призначення розділених збірок - бути бібліотеками загального використання. Оскільки будь-який додаток може отримати доступ до збірки, то виникають питання безпеки, а саме такі питання:

Колізія імен (можливий такий варіант що ваша збірка використовує типи з тими ж іменами, що використовуються у розділеній збірці).

Питання сумісності.

Вирішенням цих проблем є розміщення розділених збірок у піддереві файлової системи (GAC - global assembly cache) - глобальному кеші збірок. У глобальний кеш розділених збірок, потрібно інсталювати, а не просто копіювати як у випадку з файловою системою.

ЛІТЕРАЛИ

У програмах на мовах високого рівня (у тому числі C#) літералами називають послідовність символів, що входять у алфавіт мови програмування, що забезпечують явне представлення значень, які використовуються для позначення початкових значень в оголошенні членів класів, змінних і констант в методах класу. Розрізняються літерали арифметичні (різних типів), логічні, символьні (включаючи Escape-послідовності), рядкові.

АРИФМЕТИЧНІ ЛІТЕРАЛИ

Арифметичні літерали кодують значення різних (арифметичних) типів. Тип арифметичного літерала визначається наступними інтуїтивно зрозумілими зовнішніми ознаками:

стандартним зовнішнім виглядом. Значення цілочисельного типу зазвичай кодується інтуїтивно зрозумілою послідовністю символів '1' ..., '9', '0'. Значення плаваючого типу також передбачає стандартний вигляд (крапка-роздільник між цілою і дробовою частиною, або наукова або експоненціальна нотація - 1.2500E+052). Шістнадцяткове представлення цілочисельного значення кодується шістнадцятковим літералом, що складається з символів '0' ..., '9', а також 'a' ..., 'f', або 'A' ..., 'F' з префіксом '0x';

власне значенням. 32768 ніяк не може бути значенням типа short;

додатковим суфіксом. Суфікси l, L відповідають типові long; ul, UL - unsigned long; f, F - float; d, D - decimal. Значення типа double кодуються без префікса.

ЛОГІЧНІ ЛІТЕРАЛИ

До логічних літералів відносяться наступні послідовності символів: true і false. Більше логічних літералів в C# немає.

СИМВОЛЬНІ ЛІТЕРАЛИ

Це взяті в одинарні лапки одиничні символи, що вводяться з клавіатури: 'X', 'p', 'Q', '7', а також цілочисельні значення в діапазоні від 0 до 65535, перед якими розташовується конструкція вигляду (char) - операція явного приведення до типа char: (char)34 - "", (char) 44 - ",, (char) 7541 - який символ буде тут - не ясно.

Наступні взяті в одинарні лапки послідовності символів є Escape-послідовностями. Ця категорія літералів використовується для створення додаткових ефектів (дзвінок), простого форматування інформації, що виводиться, і кодування символів при виводі і порівнянні (у виразах порівняння).

Таблиця 6

Символьні Escape-послідовності

Ім’я типу	Системний тип
\a	Звуковий сигнал
\b	Повернення на одну позицію назад
\f	Перехід на нову сторінку
\n	Перехід на новий рядок
\r	Повернення каретки
\t	Горизонтальна табуляція
\v	Вертикальна табуляція
\0	Нуль
\’	Одинарна лапка
\”	Подвійна лапка
\\	Зворотна коса лінія

Рядкові літерали - це послідовність символів і символьних Escape-послідовностей, взятих у подвійні лапки.

Verbtim string - рядковий літерал, що інтерпретується компілятором так, як він записаний. Escape-послідовності сприймаються строго як послідовності символів.

Verbatim string представляється за допомогою символу @, який розташовується безпосередньо перед строковим літералом, взятим в парні подвійні лапки. Представлення подвійних лапок в Verbatim string забезпечується їх дублюванням. Пара літералів (другий - Verbatim string)

..."c:\\My Documents\\sample.txt"...

...@"c:\My Documents\sample.txt"...

мають одне і те ж значення: c:\My Documents\sample.txt .

Представлення подвійних лапок всередині Verbatim string досягається за рахунок їх дублювання: ...@""Focus"" і має значення "Focus".

Рядкові літерали є літералами типу string.

ПРИВЕДЕННЯ ТИПІВ ДАНИХ

Приведення типів - один з аспектів безпеки мови.

Використовувані в програмі типи характеризуються власними діапазонами значень, які визначаються властивостями типів, - у тому числі і розміром області пам'яті, призначеної для кодування значень відповідного типу. При цьому області значень різних типів перетинаються. Багато значень можна виразити більш ніж одним типом. Наприклад, значення 4 можна представити як значення типа sbyte, byte, short, ushort, int, uint, long, ulong. При цьому в програмі все повинно бути влаштовано так, щоб логіка перетворення значень одного типу до іншого типу була зрозумілою, а результати цих перетворень - передбачувані.

Інколи приведення значення до іншого типа відбувається автоматично. Такі перетворення називаються неявними.

Але у ряді випадків перетворення вимагає додаткової уваги з боку програміста, який повинен явним чином вказувати необхідність перетворення, використовуючи вирази приведення типа або звертаючись до спеціальних методів перетворення, визначеним в класі System.Convert, які забезпечують перетворення значення одного типа до значення іншого.

Перетворення типу створює значення нового типу, еквівалентне значенню старого типа, проте при цьому не обов'язково зберігається ідентичність (або точні значення) двох об'єктів.

Розрізняють розширене та звужуюче перетворення:

Розширене перетворення - значення одного типу перетвориться до значення іншого типу, яке має такий же або більший розмір. Наприклад, значення, представлене у вигляді 32-розрядного цілого числа із знаком, може бути перетворене в 64-розрядне ціле число із знаком. Розширене перетворення вважається безпечним, оскільки вихідна інформація при такому перетворенні не спотворюється.

Можливість розширеного перетворення представлено у таблиці.

Можливість розширеного перетворення типів даних

Тип	У який тип перетворюється
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Float, Double, Decimal
SByte	Int16, Int32, Int64, Float, Double, Decimal
Int16	Int32, Int64, Float, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Float, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Float, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Float	Double

Звужуюче перетворення - значення одного типу перетвориться до значення іншого типа, яке має менший розмір (з 64-розрядного в 32-розрядне). Таке перетворення потенційне небезпечно втратою значення. Звужуючі перетворення можуть призводити до втрати інформації. Якщо тип, до якого здійснюється перетворення, не може правильно передати значення джерела, то результат перетворення виявляється дорівнює константі PositiveInfinity або NegativeInfinity.При цьому значення PositiveInfinity інтерпретується як результат ділення додатнього числа на нуль, а значення NegativeInfinity - як результат ділення відємного числа на нуль. Якщо звужуюче перетворення забезпечується методами класу System.Convert, то втрата інформації супроводиться генерацією виключення (про виключення пізніше).

ЛЕКЦІЯ 8.
ПОНЯТТЯ СТЕКУ І КУПІ. ЗНАЧИМІ ТА ПОСИЛАЛЬНІ ТИПИ ДАНИХ У .NET І C#.

ПОНЯТТЯ СТЕКУ І КУПІ

Стек відноситься до області пам'яті, підтримуваної процесором, в якій зберігаються локальні змінні. Доступ до стека у багато разів швидший, ніж до загальної області пам'яті, тому використання стека для зберігання даних прискорює роботу вашої програми. У C# розмірні типи (наприклад, цілі числа) розташовуються в стеку: для їх значень зарезервована область в стеку, і доступ до неї здійснюється по назві змінної.

Посилальні типи (наприклад, об'єкти) розташовуються в купі. Купа — це оперативна пам'ять вашого комп'ютера. Доступ до неї здійснюється повільніше, ніж до стека. Коли об'єкт розташовується в купі, то змінна зберігає лише адресу об'єкту. Ця адреса зберігається в стеку. За адресою програма має доступ до самого об'єкту, всі дані якого зберігаються в загальній пам'яті (купі).

«Збиральник сміття» (Garbage Collector) знищує об'єкти, розташовані в стеку, кожен раз, коли відповідна змінна виходить за область видимості. Таким чином, якщо ви оголошуєте локальну змінну в межах функції, то об'єкт буде помічений як об'єкт для «збирання сміття». І він буде видалений з пам'яті після завершення роботи функції. Об'єкти у купі теж очищаються збиральником сміття, після того, як кінцеве посилання на них буде знищено.

ЗНАЧИМІ ТА ПОСИЛАЛЬНІ ТИПИ ДАНИХ У .NET І C#

Усі типи даних у C# поділяються на значимі та типи-посилання.

Значимі типи:

містять у собі об’єкти даних;
не можуть бути пустими.

Типи-посилання:

містять у собі посилання на об’єкт даних;
можуть бути пустими (null).

Запишемо простий приклад:

```
int a = 1000;
int b = a;
b = 2000;
```

Логічно, що після виконання цього коду a = 1000, b = 2000 - це принцип роботи значимих типів - у змінну копіюється значення і не привязується до змінної з якої воно було взяте.

Тепер створимо просту структуру Point.

Використання структури (значимий тип).

```
    static void Main(string[] args)
    {
        Point point1 = new Point();
        point1.x = 111;
        point1.y = 111;

       //Виведемо на екран змінну point1
        Console.WriteLine("point1.x = {0}; point1.y = {1}", point1.x, point1.y);

// Створимо змінну point2
        Point point2 = point1;

//Виведемо на екран змінну point2
        Console.WriteLine("point2.x = {0}; point2.y = {1}", point2.x, point2.y);

Console.WriteLine("\t---- Change point 2-----");
        //Змінимо значення параметрів у point2
        point2.x = 777;
        point2.y = 777;
//Виведемо point1 та point2
        Console.WriteLine("point1.x = {0}; point1.y = {1}", point1.x, point1.y);
        Console.WriteLine("point2.x = {0}; point2.y = {1}", point2.x, point2.y);
    }
    struct Point
    {
        public int x;
        public int y;
    }
```

Рис. Результат виконання прикладу (Значимий тип).

Як бачимо після зміни значень x та y для змінної point2 значення змінної point1 не змінилося! Все правильно, адже структура є значимим типом і тому відбувається копіювання значень при присвоєнні.

Тепер давайте замінимо структуру Point на клас.

Використання класу (посилальний тип).

```
    class Point
    {
        public int x;
        public int y;
    }
```

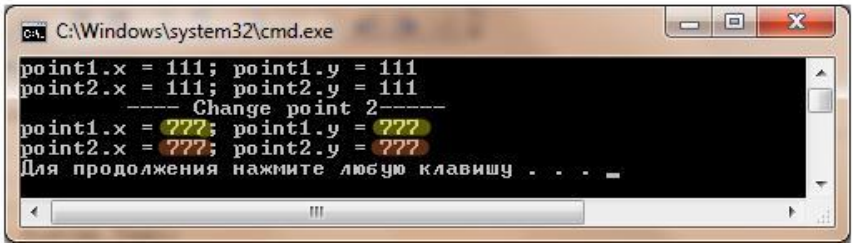


Рис. Результат виконання прикладу (Посилальний тип).

Як бачимо значення point1 змінилося! Посилальні типи даних вказують лише на ділянку памяті, у якій знаходиться значення, а не на саме значення. При присвоєнні не відбувається копіювання значень, а копіювання посилання.

Відповідно до назв, змінна в разі використання типів-значень містить власне значення, а при використанні типів-посилань – не саме значення, а лише посилання на нього. Місцем зберігання змінної, визначеної як тип-значення, є стек, а визначеною як посилальний тип – «купа» (останнє необхідне для динамічного виділення і звільнення пам'яті для зберігання змінної довільним чином).

Значенням, яким змінна ініціалізувалася за умовчанням (необхідність виконання цієї вимоги диктується ідеологією безпеки Microsoft .NET) в разі визначення за допомогою типа-значення є **0** (для чисельного типу даних), **false** (для логічного типу даних), '**\0**' (для символьного типу даних), а в разі визначення за допомогою типу-посилання – значення порожнього посилання **null**.

ЛЕКЦІЯ 9.
ОПЕРАТОРИ РОЗГАЛУЖЕННЯ C#

Оператори розгалуження

ІНСТРУКЦІЯ IF-ELSE

Формат інструкції має вигляд:

```
if(умова)
{
    інструкція;
}
else
{
    інструкція;
}
```

Тут під елементом *інструкція* розуміється одна інструкція мови C#. Частина else необов'язкова.

Якщо елемент “умова”, який є умовним виразом, при визначенні поверне значення ІСТИНА, буде виконана if-інструкція; у іншому випадку - else-інструкциія (якщо така існує). Одночасно обидві інструкції ніколи не виконуються. Умовний вираз, що управляє виконанням if-інструкції повинен мати тип bool.

Розглянемо просту програму, в якій використовується if-else-інструкція для визначення того, є число позитивним або негативним.

Перевірка значення.

```
static void Main(string[] args)
{

    double a = Console.ReadLine();

    if(a < 0)
        Console.WriteLine("a < 0");

    else
        Console.WriteLine("a >= 0");

}
```

Кострукції if-else можуть бути вкладені, причому вкладення можуть бути багатьох рівнів. Для прикладу напишемо програмку яка перевіряє чи число позитивне чи негативне і при цьому потрапляє у діапазон від -100 до 100.

Перевірка значення.

```
static void Main(string[] args)
{
    Console.Write("Введіть значення:\t");
    double a = Convert.ToDouble(Console.ReadLine());
    if (a > -100 && a < 100)
    {
        if (a < 0)
            Console.WriteLine("a < 0");
        else

            Console.WriteLine("a >= 0");
     }
     else
         Console.WriteLine("a is out of range (-100; 100) - {0}", a);
 }
```

КОНСТРУКЦІЯ ELSE-IF-ELSE.

Другою інструкцією вибору є switch. Інструкція switch забезпечує багатонаправлене розгалуження. Вона дозволяє робити вибір однієї з множини альтернатив. Хоча багатонаправлене тестування можна реалізувати з допомогою послідовності вкладених if-інструкцій, для багатьох ситуацій інструкція switch виявляється ефективнішим рішенням. Вона працює наступним чином:

1. Значення виразу послідовно порівнюється з константами із заданого списку.

2. При виявленні збігу для однієї з умов порівняння виконується послідовність інструкцій,

пов'язана з цією умовою.

Часто у програмуванні для перевірки кількох умов використовується конструкція if-else-if:

```
if(умова)
{}
else
    if(умова)
    {}
    else
        if(умова)
        {}
        ...
         else {}
```

Умовні вирази обчислюються зверху вниз. Як тільки в якій-небудь гілці виявиться істинний результат, буде виконана інструкція, пов'язана з цією гілкою, а всі інші сходинки пропускаються. Якщо виявиться, що жодна з умов не є істинною, буде виконана остання else-інструкція (можна вважати, що вона виконує роль умови за замовчуванням. Якщо остання else-інструкція не задана, а всі інші виявилися помилковими, то взагалі жодна дія не буде виконана.

ІНСТРУКЦІЯ SWITCH

Загальний формат запису інструкції switch такий:

Загальний вигляд інструкції switch.

```
switch (вираз)
{

    case костанта1:
        інструкція;

        break;

    case костанта2:
        інструкція;

        break;
    ...

     default:
        інструкція;

        break;
}
```

Елемент вираз інструкції switch повинен мати цілочисельний тип (наприклад, char, byte, short або int) або тип string. Вирази, що мають типи з плаваючою крапкою, не дозволені. Дуже часто як switch-вирази використовується просто змінна; case-константи мають бути літералами, тип яких сумісний з типом заданого виразу. При цьому жодні дві case-константи в одній switch-інструкції не можуть мати ідентичних значень.

Послідовність інструкцій default-гілки виконується в тому випадку, якщо жодна із заданих case-констант не збіжиться з результатом обчислення switch-виразу. Гілка default необов'язкова. Якщо вона відсутня, то при неспівпаданні результату виразу ні з однією з case-констант жодна дія виконана не буде. Якщо такий збіг все-таки станеться, будуть виконані інструкції, відповідні даній case-гілці до тих пір, поки не зустрінеться інструкція break.

Використання switch-інструкції демонструється у наступній програмі.

Використання switch.

```
static void Main(string[] args)
{
    Console.WriteLine(" Введіть цифру від 1 до 5.");
    int a = Convert.ToInt32(Console.ReadLine());
    switch (a)
    {
        case 1:
            Console.WriteLine("Ви ввели 1");
            break;
         case 2:
```

```
            Console.WriteLine("Ви ввели 2");
            break;
        case 3:
            Console.WriteLine("Ви ввели 3");
            break;
        case 4:
            Console.WriteLine("Ви ввели 4");
            break;
        case 5:
            Console.WriteLine("Ви ввели 5");
            break;
        default:
            Console.WriteLine(" Ви промахнулися");
            break;
    }
}
```

У випадку, якщо потрібно щоб одна і та ж сама інструкція виконалася для кількох констант одразу використовується наступна конструкція.

Використання switch.

```
static void Main(string[] args)
{
    Console.WriteLine(" Введіть цифру від 1 до 5.");
    int a = Convert.ToInt32(Console.ReadLine());

    switch (a)
    {
        case 1:
        case 2:
        case 3:
            Console.WriteLine("Ви ввели 1 або 2 або 3");
            break;
        case 4:
            Console.WriteLine("Ви ввели 4");
            break;
        case 5:
            Console.WriteLine("Ви ввели 5");
            break;
        default:
            Console.WriteLine(" Ви промахнулися");
            break;
    }
}
```

ТЕРНАРНИЙ ОПЕРАТОР

Одним з хороших операторів C# є тернарный оператор ?:. Оператор ?: часто використовується для заміни певних типів конструкцій if-else. Оператор ?: називається тернарним, оскільки він працює з трьома виразами. Його загальний формат запису має такий вигляд:

Вираз1 **?** Вираз2 **:** Вираз3 **;**

Тут Вираз1 повинен мати тип bool. Типи елементів Вираз2 і Вираз3 мають бути однакові. Зверніть увагу на використання і розміщення двокрапки.

Значення ?:-вираз визначається таким чином. Обчислюється Вираз1. Якщо він виявляється істинним, обчислюється Вираз2, і результат його обчислення стає значенням всього ?-виразу. Якщо результат обчислення елементу Вираз1 виявляється помилковим, значенням всього ?-виразу стає результат обчислення елементу Вираз3. Розглянемо приклад, в якому змінні absval присвоюється значення змінної a по модулю.

Тернарний оператор. Знаходження модуля числа.

```
static void Main(string[] args)
{
    Console.Write("Введіть цілу цифру - ");
    int a = Convert.ToInt32(Console.ReadLine());
    int absval = (a > 0) ? a : -a;
    Console.WriteLine("ABS: {0}", absval);
}
```

Щоб знайти модуль числа можна скористатися функцією Math.Abs(значення):

int absval = Math.Abs(a).

ЛЕКЦІЯ 10.
ЦИКЛИ C#

Оператори циклу

У C# існують 4 оператори циклу:

1. for
2. while
3. do-while
4. foreach

ЦИКЛ FOR

Загальний формат запису циклу for для повторного виконання однієї інструкції має наступний вигляд:

```
for (ініціалізація; умова; ітерація)
    {
        інструкція;
    }
```

Знайти суму усіх цілих чисел від 1 до 10 включно:

```
int sum = 0;
for (int i = 1; i <= 10; ++i)
      sum += i;
```

Знайти суму та добуток чисел від 1 до 10 включно:

```
int sum = 0;
int dob = 1;
for (int i = 1; i <= 10; ++i)
{
    sum += i;
    dob *= i;
}
```

Елемент ініціалізація зазвичай є інструкцією присвоєння яка встановлює значення змінної, що управляє циклом. Ця змінна працює як лічильник, який управляє роботою циклу.

Елемент умова є виразом типу bool, в якому тестується значення змінної, що управляє циклом. Результат цього тестування визначає виконається цикл for ще раз чи ні. Елемент ітерація - це вираз, який визначає, як змінюється значення змінної, що управляє циклом, після кожної ітерації. Зверніть увагу на те, що всі ці елементи циклу for повинні розділятися крапкою з комою.

Цикл for виконуватиметься до тих пір, поки обчислення елементу умова дає істинний результат. Як тільки умова стане хибною, виконання програми продовжиться з інструкції, наступної за циклом for.

Змінна, що управляє циклом, може змінюватися як з позитивним, так і з негативним приростом, причому величина цього приросту також може бути будь-якою. Наприклад, наступна програма виводить числа в діапазоні від 50 до - 50 з декрементом, рівним 5.

```
For (int i = 50; i > -50; i-=5)
    Console.WriteLine(i);
```

У циклі for управляючих змінних, умов або ітерацій може бути кілька, або може не бути взагалі.

```
static void Main(string[] args)
{
    For (int i = 0, j = 2; i < 10 && j < 5; j++, i+=2)
        Console.WriteLine("i = {0}, j = {1}", i, j);
}
```

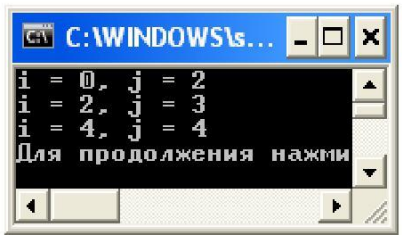


Рис. Результат виконання.

```
static void Main(string[] args)
{
    for (int i = 0; ; )
    {
        i++;
        Console.Write(" {0}  ", i);
        if (i == 5)
            break;
    }
}
```

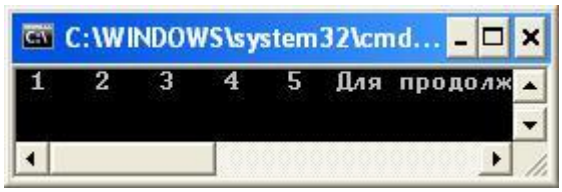


Рис. Результат виконання.

Також існує варіант коли цикл може не мати тіла. Таким чином усі операції виконуються у блоці ітерація. Розв'яжемо ту ж саму задачу про знаходження суми від 1 до 10.

```
int sum = 0;
for (int i = 1; i <= 10; sum += i++);
```

ЦИКЛ WHILE

Цикл while є універсальним виглядом циклу, що включається у всі мови програмування. Тіло циклу виконується до тих пір, поки залишається істинною умова while. У мові C# у циклу while дві модифікації - з перевіркою умови на початку і в кінці циклу. Перша модифікація має наступний синтаксис:

While (вираз) інструкції;

Ця модифікація відповідає стратегії: "спочатку перевір, а потім роби". В результаті перевірки може виявитися, що і робити нічого не потрібно. Тіло такого циклу може жодного разу не виконуватися. Звичайно ж, можливо і зациклення. У нормальній ситуації кожне виконання тіла циклу - це черговий крок до завершення циклу.

Цикл, перевіряючий умову завершення в кінці, відповідає стратегії: "спочатку роби, а потім перевір". Тіло такого циклу виконується, щонайменше, один раз. Ось синтаксис цієї модифікації:

```
do
{
    інструкція;
}
While (вираз);
```

Роботу циклу foreach буде розглянуто детальніше у наступній темі (робота з масивами).

ЛЕКЦІЯ 11.
ОСНОВИ РОБОТИ З МАСИВАМИ

Поняття масиву даних

Масив задає спосіб організації даних. ***Масивом*** називають впорядковану сукупність елементів одного типу. Кожен елемент масиву має індекси, що визначають порядок елементів. Число індексів характеризує розмір масиву. Кожен індекс змінюється в деякому діапазоні [a,b]. У мові C#, як і в багатьох інших мовах, індекси задаються цілочисельним типом. Діапазон [a,b] називається граничною парою, a – нижньою межою, b – верхньою межею. Якщо межі задані константними виразами, то число елементів масиву відоме у момент його оголошення і йому може бути виділена пам'ять ще на етапі трансляції. Такі масиви називаються статичними. Якщо ж вирази, які задають межі, залежать від змінних, то такі масиви називаються динамічними, оскільки пам'ять їм може бути відведена лише динамічно в процесі виконання програми, коли стають відомими значення відповідних змінних. Масиву, як правило, виділяється безперервна область пам'яті.

У мові C++ всі масиви є статичними.

У мові C# знято істотне обмеження мови C++ на статичність масивів. Масиви в мові C# є справжніми динамічними масивами. Як наслідок цього масиви відносяться до посилальних типів, пам'ять їм відводиться динамічно в "купі".

У мові C++ "класичних" багатовимірних масивів немає. Тут введені одновимірні масиви і масиви масивів. Останні є загальнішою структурою даних і дозволяють задати не лише багатовимірний куб, але і порізану, ступінчасту структуру.

У мові C# збережені одновимірні масиви і масиви масивів. На додаток до них в мову додані багатовимірні масиви. Динамічні багатовимірні масиви мови C# є потужною, надійною, зрозумілою і зручною структурою даних, яку сміливо можна рекомендувати до вживання не лише професіоналам, але і новачкам, що програмують на C#. Після цього короткого огляду давайте перейдемо до більш систематичного вивчення деталей роботи з масивами в C#.

Одновимірні масиви

ОГОЛОШЕННЯ ОДНОВИМІРНИХ МАСИВІВ

У спрощеному вигляді оголошення одновимірного масиву виглядає таким чином:

тип[] ім’я_змінної;

Увага, на відміну від мови C++ квадратні дужки приписані не до імені змінної, а до типу. Вони є невід'ємною частиною визначення класу, так що запис T[] слід розуміти як клас одновимірний масив з елементами типу T.

Що ж до меж зміни індексів, то ця характеристика до класу не відноситься, вона є характеристикою змінних - екземплярів, кожен з яких є одновимірним масивом зі своїм числом елементів, що задаються в оголошенні змінної.

int[] a, b, c;

Найчастіше при оголошенні масиву використовується ім'я з ініціалізацією. І знову-таки, як і в разі простих змінних, можуть бути два варіанти ініціалізації. У першому випадку ініціалізація є явною і задається константним масивом. Ось приклад:

double[] x= {5.5, 6.6, 7.7};

Слідуючи синтаксису, елементи константного масиву слід брати у фігурні дужки.

У другому випадку створення і ініціалізація масиву виконується в об'єктному стилі з викликом конструктора масиву. І це найбільш поширена практика оголошення масивів. Наведу приклад:

int[] d= new int[5];

Отже, якщо масив оголошується без ініціалізації, то створюється лише висяче посилання із значенням void. Якщо ініціалізація виконується конструктором, то в динамічній пам'яті створюється сам масив, елементи якого ініціалізувалися константами відповідного типу, і посилання зв'язується з цим масивом. Якщо масив ініціалізувався константним масивом, то в пам'яті створюється константний масив, з яким і зв'язується посилання.

ІНІЦІАЛІЗАЦІЯ МАСИВІВ

Ініціалізувати масиви, наприклад, можна наступними способами:

1. Літералами відповідного типу
2. Випадковими числами
3. Ввести з клавіатури

Заповнення масиву літералами може відбуватися наступним чином:

Заповнення масиву літералами.

```
int[] nums = { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };
```

```
int[] nums;
nums = new int[ ] { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };

int[] nums = new int[10] { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };
```

Приклад програми на C#, яка заповнює масив випадковими числами:

"Заповнення масиву випадковими числами".

```
static void Main(string[] args)
{

    int[] array = new int[10];

    Random rand = new Random();
    for (int i = 0; i < 10; ++i)
    {
        array[i] = rand.Next(0, 100);
```

```
1:              Console.Write(" {0}", array[i]);
2:          }

3:      }
```

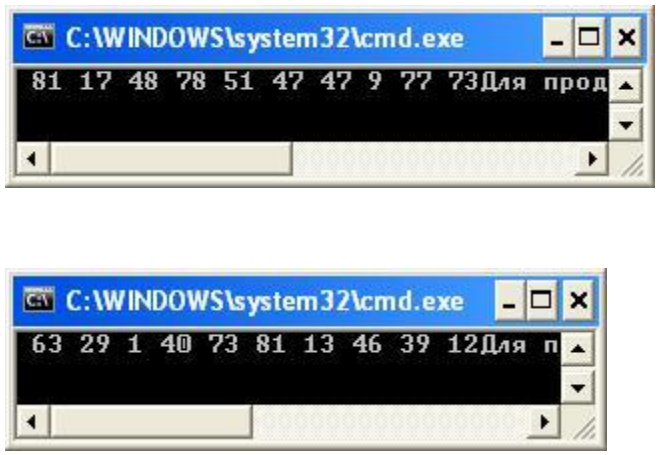


Рис. Результат виконання прикладу.

Ввести з клавіатури значення масиву можна наступним чином:

"Заповнення масиву значеннями, введеними з клавіатури".

```
        static void Main(string[] args)

        {
            int[] array;
            Console.Write("\n Введіть розмір масиву >\t");
            int size = Convert.ToInt32(Console.ReadLine());
            array = new int[size];


            for (int i = 0; i < array.Length; ++i)
            {
                Console.Write("Введіть {0}-й елемент:\t", i+1);
                array[i] = Convert.ToInt32(Console.ReadLine());
            }
        }
```

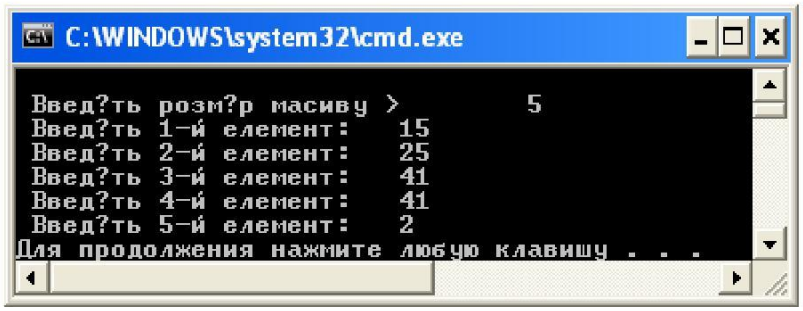


Рис. Результат виконання прикладу.

ЛЕКЦІЯ 12.
БАГАТОВИМІРНІ МАСИВИ

Жодної принципової різниці між одновимірними та багатовимірними масивами немає. Одновимірні масиви - це окремий випадок багатовимірних. Можна говорити і по-іншому: багатовимірні масиви є природним узагальненням одновимірних. Одновимірні масиви дозволяють задавати такі математичні структури як вектори, двовимірні, - матриці, тривимірні, - куби даних, масиви більшої розмірності - багатовимірні куби даних. Відмічу, що при роботі з базами даних багатовимірні куби, так звані куби OLAP, зустрічаються часто.

У чому особливість оголошення багатовимірного масиву? Як в вказати розмірність масиву? Це робиться досить просто, за рахунок використання ком. От як виглядає оголошення багатовимірного масиву в загальному випадку:

```
int[,] array;
```

Приклад ініціалізації:

```
int[,] matrix = {1,2},{3,4};
```

Давайте розглянемо класичне завдання додавання прямокутних матриць. Нам знадобиться три динамічні масиви для представлення матриць і три методи, один з яких заповнюватиме вхідні матриці випадковими числами, інший - виконувати додавання матриць, третій - друкувати самі матриці.

Ось тестовий приклад (програма містить трішки довгий код, але варто його переглянути):

Додавання матриць.

```
class AddMatrixClass
{
    static void Main(string[] args)
    {
        //Оголосимо розмірність наших матриць
        int rows = 3, cols = 5;

        //Оголосимо матриці
        int[,] a = new int[rows, cols];
        int[,] b = new int[rows, cols];
        int[,] result = new int[rows, cols];

        //Заповнимо матриці випадковими числами
        a = FillArray(rows, cols, 5);
        b = FillArray(rows, cols, 7);
        //Виведемо на консоль матрицю A
        Console.WriteLine("\nArray1: ");
        PrintArray(a);
        //Виведемо на консоль матрицю B
        Console.WriteLine("\nArray2: ");
        PrintArray(b);
```

```
            //Додамо і присвоїмо результат
            result = AddMatrix(a, b);

            if (result != null)
            {
                Console.WriteLine("\nResultMatrix: ");
                PrintArray(result);
            }
        }

        static int[,] AddMatrix(int[,] a, int[,] b)
        {
            //Оголосимо матрицю у яку будемо записувати результат.
            int[,] res = new int[a.GetLength(0), a.GetLength(1)];

            //Перевіримо чи однаковий розмір матриць
            if ((a.GetLength(0) != b.GetLength(0)) || (a.GetLength(1) != b.GetLength(1)))
                Console.WriteLine("Мариці неоднакового розміру.");
            else
            {
                for (int i = 0; i < a.GetLength(0); ++i)
                {
                    //Виконуємо додавання елементів матриць
                    for (int j = 0;   j < a.GetLength(1); ++j)
                        res[i, j] =   a[i, j] + b[i, j];
                }
                return res;
            }
            return null;
        }

        static int[,] FillArray(int rows, int cols, int randnum)
        {
            int[,] array = new int[rows, cols];
            //Створює змінну класу рендом для генерування значень
           Random rand = new Random(randnum);
            for (int i = 0; i < rows; ++i)
            {
                for (int j = 0; j < cols; ++j)
                {
            array[i, j] = rand.Next(0,100);
                }
            }
            return array;
        }

        static void PrintArray(int[,] array)
        {
            for (int i = 0; i < array.GetLength(0); ++i)
            {
                for (int j = 0; j < array.GetLength(1); ++j)
                {
//Виводимо значення на консоль
```

```
Console.Write(" {0}\t", array[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
```

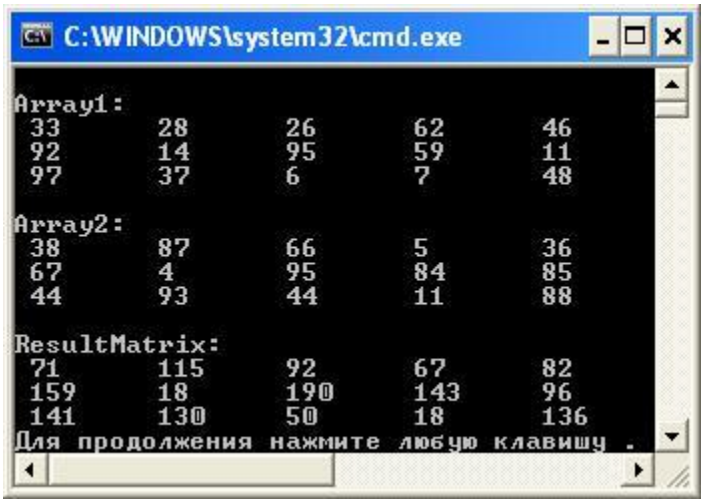


Рис. Результат виконання прикладу.

Масиви масивів

Ще одним видом масивів C# є масиви масивів, звані також порізаними/рваними масивами (jagged arrays). Такий масив масивів можна розглядати як одновимірний масив, елементи якого є масивами, елементи яких, у свою чергу, знову можуть бути масивами, і так може тривати до деякого рівня вкладеності.

У яких ситуаціях може виникати необхідність в таких структурах даних? Ці масиви можуть застосовуватися для представлення дерев, в яких вузли можуть мати довільне число нащадків. Таким може бути, наприклад, генеалогічне дерево. Вершини першого рівня - Fathers, що представляють батьків, можуть задаватися одновимірним масивом, так що Fathers[i] - це і-й батько. Вершини другого рівня представляються масивом масивів - Children, так що Children[i] - це масив дітей і-го батька, а Children[i][j] - це j-й дитя і-го батька. Для представлення внуків знадобиться третій рівень, так що GrandChildren [i][j][k] представлятиме k-го внука j-го дитяти і-го батька.

Є деякі особливості в оголошенні і ініціалізації таких масивів. Якщо при оголошенні багатовимірних масивів для вказівки розмірності використовувалися коми, то для порізаних масивів застосовується ясніша символіка - сукупності пар квадратних дужок; наприклад, int[][] задає масив, елементи якого - одновимірні масиви елементів типа int.

Складніше із створенням самих масивів і їх ініціалізацією. Тут не можна викликати конструктор new int[3][5], оскільки він не задає порізаний масив. Фактично потрібно викликати конструктор для кожного масиву на самому нижньому рівні. У цьому і полягає складність оголошення таких масивів. Почну з формального прикладу:

```
int[][] jagger = new int[3][]
{
    new int[] {5,7,9,11},
```

```
        new int[] {2,8},
        new int[] {6,12,4}
};
```

Масив jagger має всього два рівні. Можна вважати, що у нього три елементи, кожен з яких є масивом. Для кожного такого масиву необхідно викликати конструктор new, аби створити внутрішній масив. У даному прикладі елементи внутрішніх масивів набувають значення, будучи явно ініціалізовані константними масивами. Звичайно, допустимим є і таке оголошення:

```
 int[][] jagger1 = new int[3][]
{
        new int[4],
        new int[2],
        new int[3]
};
```

В цьому випадку елементи масиву набудуть при ініціалізації нульових значень. Реальну ініціалізацію потрібно буде виконувати програмним шляхом. Варто відмітити, що в конструкторі верхнього рівня константу 3 можна опустити і писати просто new int[][]. Найзабавніше, що виклик цього конструктора можна взагалі опустити - він матиметься на увазі:

```
int[][] jagger2 =
{
        new int[4],
        new int[2],
        new int[3]
};
```

Оголошувати вкладені масиви обов’язково.

Цикл foreach та масиви

У лекції про цикли загдувалося, що у мові C# визначений цикл foreach, але детальний його розгляд був відкладений «на потім». Час для нього настав.

Цикл foreach використовується для опиту елементів колекції. Колекція – це група об'єктів. C# визначає декілька типів колекцій, і одним з них є масив. Формат запису циклу foreach має такий вигляд:

foreach (тип ім'я_змінної in колекція) інструкція;

Тут елементи тип та ім’я_змінної задають тип та ім'я ітераційної змінної, яка при функціонуванні циклу foreach набуватиме значень елементів з колекції. Елемент колекція служить для вказівки опитуваної колекції (в даному випадку як колекцію ми розглядаємо масив). Таким чином, елемент тип повинен збігатися (або бути сумісним) з базовим типом масиву. Тут важливо запам'ятати, що ітераційну змінну стосовно масиву можна використовувати лише для читання. Отже, неможлив змінити вміст масиву, присвоївши ітераційній змінній нове значення.

Розглянемо простий приклад використання циклу foreach. Приведена нижче програма створює масив для зберігання цілих чисел і присвоює його елементам початкові значення. Потім вона відображає елементи масиву, попутно обчислюючи їх суму.

Робота з циклом foreach.

```
static void Main(string[] args)
{
    int[] array = new int[10];
    int suma = 0;
    array = FillArrayRandom(10);
    Console.Write("Array:\t");
    PrintArray(array);
    foreach (int element in array)
    {
        Console.Write("{0} ", element);
        suma += element;
    }
    Console.WriteLine("\n\nСума едементів:\t{0}", suma);
}
```

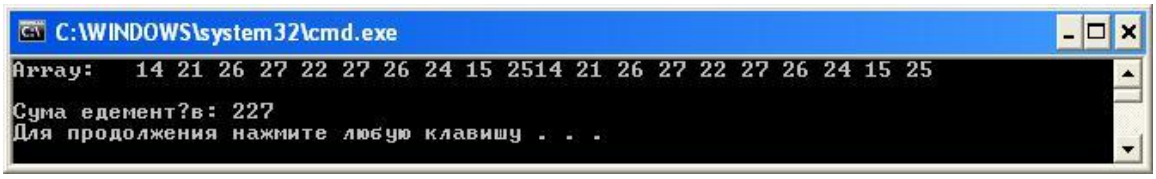


Рис. Результат виконання прикладу.

ЛЕКЦІЯ 13.
Операції з рядками

Конкатенація

Конкатенация строк або об'єднання може проводитися як за допомогою операції +, так і за допомогою методу Concat:

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + "" + s2; // Результат: рядок "hello world"
string s4 = String.Concat (s3, "!!!"); // Результат: рядок "hello world !!!"
Console.WriteLine (s4);
```

Метод Concat є статичним методом класу String, які приймають в якості параметрів два рядки. Також є інші версії методу, які беруть іншу кількість параметрів.
Для об'єднання рядків також може використовуватися метод **Join**:

```
string s5 = "apple";
string s6 = "a day";
string s7 = "keeps";
string s8 = "a doctor";
string s9 = "away";
string [] values = new string [] {s5, s6, s7, s8, s9};
String s10 = String.Join ( "", values);
// Результат: рядок "apple a day keeps a doctor away"
```

Метод Join також є статичним. Використана вище версія методу отримує два параметри: рядок-роздільник (в даному випадку пробіл) і масив рядків, які будуть з'єднуватися і розділятися роздільником.

Порівняння рядків

Для порівняння рядків застосовується статичний метод **Compare**:

```
string s1 = "hello";
string s2 = "world";
int result = String.Compare (s1, s2);
if (result <0)
{
   Console.WriteLine ( "Рядок s1 перед рядком s2");
}
else if (result> 0)
{
   Console.WriteLine ( "Рядок s1 варто після рядка s2");
}
else
{
   Console.WriteLine ( "Рядки s1 і s2 ідентичні");
}
// Результатом буде "Рядок s1 перед рядком s2"
```

Дана версія методу Compare приймає два рядки і повертає число. Якщо перший рядок за алфавітом

стоїть вище другий, то повертається число менше нуля. В іншому випадку повертається число більше нуля. І третій випадок - якщо рядки рівні, то повертається число 0.
В даному випадку так як символ h за алфавітом стоїть вище символу w, то і перший рядок буде стояти вище.

Пошук в рядку

За допомогою методу **IndexOf** ми можемо визначити індекс першого входження окремого символу або підрядка в рядку:

```
string s1 = "hello world";
char ch = 'o';
int indexOfChar = s1.IndexOf (ch); // Одно 4
Console.WriteLine (indexOfChar);
string subString = "wor";
int indexOfSubstring = s1.IndexOf (subString); // Одно 6
Console.WriteLine (indexOfSubstring);
```

Подібним чином діє метод **LastIndexOf**, тільки знаходить індекс останнього входження символу або підрядка в рядок.
Ще одна група методів дозволяє дізнатися починається або закінчується рядок на певну подстроку. Для цього призначені методи **StartsWith** і **EndsWith**. Наприклад, у нас є завдання видалити з папки всі файли з розширенням exe:

```
string path = @ "C: \ SomeDir";
string [] files = Directory.GetFiles (path);
for (int i = 0; i <files.Length; i ++)
{
   if (files [i] .EndsWith ( ". exe"))
      File.Delete (files [i]);
}
```

Поділ рядків

За допомогою функції **Split** ми можемо розділити рядок на масив підрядків. Як параметр функція Split приймає масив символів або рядків, які і будуть служити роздільниками. Наприклад, підрахуємо кількість слів у терміні, розділивши її з пробільним символам:

```
string text = "І тому всі так сталося";
string [] words = text.Split (new char [] { "});
foreach (string s in words)
{
   Console.WriteLine (s);
}
```

Це не кращий спосіб поділу по прогалин, так як у вхідному рядку у нас могло б бути кілька поспіль прогалин і в підсумковий масив також би потрапляли прогалини, тому краще використовувати іншу версію методу:

```
string [] words = text.Split (new char [] { "}, StringSplitOptions.RemoveEmptyEntries);
```

Другий параметр StringSplitOptions.RemoveEmptyEntries каже, що треба видалити всі порожні підрядка.

Обрізка рядків

Для обрізки початкових або кінцевих символів використовується функція **Trim**:

```
string text = "hello world";
text = text.Trim (); // Результат "hello world"
text = text.Trim (new char [] { 'd', 'h'}); // Результат "ello worl"
```

Функція Trim без параметрів обрізає початкові і кінцеві прогалини і повертає обрізану рядок. Щоб явно вказати, які початкові та кінцеві символи слід обрізати, ми можемо передати в функцію масив цих символів.
Ця функція має часткові аналоги: функція **TrimStart** обрізає початкові символи, а функція **TrimEnd** обрізає кінцеві символи.
Обрізати певну частину рядка дозволяє функція **Substring**:

```
string text = "Хороший день";
// Обрізаємо починаючи з третього символу
text = text.Substring (2);
// Результат "роший день"
Console.WriteLine (text);
// Обрізаємо спочатку до останніх двох символів
text = text.Substring (0, text.Length - 2);
// Результат "роший де"
Console.WriteLine (text);
```

Функція **Substring** також повертає обрізану рядок. Як параметр перша використана версія застосовує індекс, починаючи з якого треба обрізати рядок. Друга версія застосовує два параметра - індекс початку обрізки і довжину обрізана частини рядка.

Вставка
Для вставки одного рядка в іншу застосовується функція **Insert**:

```
string text = "Хороший день";
string subString = "чудовий";

text = text.Insert (8, subString);
Console.WriteLine (text);
```

Першим параметром в функції Insert є індекс, за яким треба вставляти подстроку, а другий параметр - власне подстрока.

Видалення рядків

Видалити частину рядка допомагає метод **Remove**:

```
string text = "Хороший день";
// Індекс останнього символу
int ind = text.Length - 1;
// Вирізаємо останній символ
text = text.Remove (ind);
Console.WriteLine (text);
// Вирізаємо перші два символи
text = text.Remove (0, 2);
```

Перша версія методу Remove приймає індекс в рядку, починаючи з якого треба видалити всі символи. Друга версія приймає ще один параметр - скільки символів треба видалити.

Заміна

Щоб замінити один символ або подстроку на іншу, застосовується метод **Replace**:

```
string text = "хороший день";
text = text.Replace ( "хороший", "поганий");
Console.WriteLine (text);
text = text.Replace ( "про", "");
Console.WriteLine (text);
```

У другому випадку застосування функції Replace рядок з одного символу "про" замінюється на порожній рядок, тобто фактично віддаляється з тексту. Подібним способом легко видаляти якийсь певний текст в рядках.

Зміна регістра
Для приведення рядка до верхнього і нижнього регістру використовуються відповідно функції **ToUpper()** і **ToLower()**:

```
string hello = "Hello world!";
Console.WriteLine (hello.ToLower ()); // Hello world!
Console.WriteLine (hello.ToUpper ()); // HELLO WORLD!
```

ЛЕКЦІЯ 14.
РОБОТА З ФОРМАМИ

Зовнішній вигляд програми є нам переважно через форми. Форми є основними будівельними блоками. Вони надають контейнер для різних елементів управління. А механізм подій дозволяє елементам форми відгукуватися на введення користувача, і, таким чином, взаємодіяти з користувачем.

При відкритті проекту в Visual Studio в графічному редакторі ми можемо побачити візуальну частину форми - ту частину, яку ми бачимо після запуску програми і куди ми переносимо елементи з панелі управління. Але насправді форма приховує потужний функціонал, що складається з методів, властивостей, подій та інше. Розглянемо основні властивості форм.

Якщо ми запустимо додаток, то нам відобразиться одна порожня форма. Однак навіть такий простий проект з порожньою формою має кілька компонентів.
Незважаючи на те, що ми бачимо тільки форму, але стартовою точкою входу в графічне додаток є клас Program, розташований в файлі Program.cs

Спочатку програмою запускається даний клас, потім за допомогою виразу Application.Run (new Form1 ()) він запускає форму Form1. Якщо раптом ми захочемо змінити стартову форму в додатку на якусь іншу, то нам треба змінити в цьому виразі Form1 на відповідний клас форми.
Сама форма складна за змістом. Вона ділиться на ряд компонентів.
Оголошується частковий клас форми Form1, яка має два методи: Dispose (), який виконує роль деструктора об'єкта, і InitializeComponent (), який встановлює початкові значення властивостей форми.
При додаванні елементів управління, наприклад, кнопок, їх опис також додається в цей файл.
Але на практиці ми рідко будемо стикатися з цим класом, так як вони виконує в основному дизайнерські функції - установка властивостей об'єктів, установка змінних.
Ще один файл - Form1.resx - зберігає ресурси форми. Як правило, ресурси використовуються для створення одноманітних форм відразу для декількох мовних культур.
І більш важливий файл - Form1.cs, який в структурі проекту називається просто Form1, містить код або програмну логіку форми.
За замовчуванням тут є тільки конструктор форми, в якому просто викликається метод InitializeComponent (), оголошений у файлі дизайнера Form1.Designer.cs.

Основні властивості форм

Більшість цих властивостей впливає на візуальне відображення форми. Стисло нагадаємо основні властивостями:

Name: встановлює ім'я форми - точніше ім'я класу, який успадковується від класу Form
BackColor: вказує на фоновий колір форми. Клацнувши на це властивість, ми зможемо вибрати той колір, який нам підходить зі списку запропонованих кольорів або кольорової палітри
BackgroundImage: вказує на фонове зображення форми
BackgroundImageLayout: визначає, як зображення, задане у властивості BackgroundImage, буде розташовуватися на формі.
ControlBox: показує, чи показується меню форми. В даному випадку під меню розуміється меню самого верхнього рівня, де знаходяться іконка програми, заголовок форми, а також кнопки мінімізації форми і хрестик. Якщо ця властивість має значення false, то ми не побачимо ні іконку, ні хрестика, за допомогою якого зазвичай закривається форма
Cursor: визначає тип курсора, який використовується на формі

Enabled: якщо ця властивість має значення false, то вона не зможе отримувати введення від користувача, тобто ми не зможемо натиснути на кнопки, ввести текст в текстові поля і т.д.
Font: задає шрифт для всієї форми і всіх поміщених на неї елементів управління. Однак, задавши у елементів форми свій шрифт, ми можемо тим самим перевизначити його
ForeColor: колір шрифту на формі
FormBorderStyle: вказує, як буде відображатися межа форми і рядок заголовка. Встановлюючи дане властивість в None можна створювати зовнішній вигляд програми довільної форми
HelpButton: показує, чи показується кнопка довідки форми
Icon: задає іконку форми
Location: визначає положення по відношенню до верхнього лівого кута екрана, якщо для властивості StartPosition встановлено значення Manual
MaximizeBox: вказує, чи буде доступна кнопка максимізації вікна в заголовку форми
MinimizeBox: вказує, чи буде доступна кнопка мінімізації вікна
MaximumSize: задає максимальний розмір форми
MinimumSize: задає мінімальний розмір форми
Opacity: задає прозорість форми
Size: визначає початковий розмір форми
StartPosition: вказує на початкову позицію, з якою форма з'являється на екрані
Text: визначає заголовок форми
TopMost: якщо ця властивість має значення true, то форма завжди буде знаходитися поверх інших вікон
Visible: видима чи форма, якщо ми хочемо приховати форму від користувача, то можемо поставити даній властивості значення false
WindowState: вказує, в якому стані форма буде знаходитися при запуску: в нормальному, максимізувати або мінімізованому

Програмна настройка властивостей

За допомогою значень властивостей у вікні Властивості ми можемо змінити на свій розсуд зовнішній вигляд форми, але все те ж саме ми можемо зробити динамічно в коді.
За допомогою спеціального вікна *Properties* (Властивості) праворуч Visual Studio надає нам зручний інтерфейс для управління властивостями елемента.

Установка розмірів форми

Для установки розмірів форми можна використовувати такі властивості як *Width / Height* або *Size*. *Width / Height* приймають числові значення, як в наведеному вище прикладі. При установці розмірів через властивість *Size*, нам треба привласнити властивості об'єкт типу *Size*:
this.Size = new Size(200,150);
Об'єкт *Size* в свою чергу приймає в конструкторі числові значення для установки ширини і висоти.

Початкове розташування форми

Початкове розташування форми встановлюється за допомогою властивості *StartPosition*, яке може приймати одне з наступних значень:

Manual: Положення форми визначається властивістю *Location*
CenterScreen: Положення форми в центрі екрана
WindowsDefaultLocation: Позиція форми на екрані задається системою Windows, а розмір визначається властивістю *Size*
WindowsDefaultBounds: Початкова позиція і розмір форми на екрані задається системою Windows

CenterParent: Положення форми встановлюється в центрі батьківського вікна

Всі ці значення містяться в перерахуванні *FormStartPosition*, тому, щоб, наприклад, встановити форму в центрі екрану, нам треба прописати так:
this.StartPosition = FormStartPosition.CenterScreen;

ЛЕКЦІЯ 15.
РОБОТА З ФОРМАМИ

Фон і колір форми

Щоб встановити колір як фону форми, так і шрифту, нам треба використовувати колірне значення, що зберігається в структурі Color:
this.BackColor = Color.Aquamarine;
this.ForeColor = Color.Red;
Крім того, ми можемо в якості фону задати зображення у властивості *BackgroundImage*, вибравши його у вікні властивостей або в коді, вказавши шлях до зображення:
this.BackgroundImage = Image.FromFile("C:\Users\Eugene\Pictures\3332.jpg");
Щоб належним чином налаштувати потрібний нам відображення фонової картинки, треба використовувати властивість *BackgroundImageLayout*, яке може приймати одне з наступних значень:

None: Зображення поміщається в верхньому лівому куті форми і зберігає свої початкові значення
Tile: Зображення розташовується на формі у вигляді мозаїки
Center: Зображення розташовується по центру форми
Stretch: Зображення розтягується до розмірів форми без збереження пропорцій
Zoom: Зображення розтягується до розмірів форми зі збереженням пропорцій

Наприклад, розташуємо форму по центру екрана:
this.StartPosition = FormStartPosition.CenterScreen;

Додавання форм. Взаємодія між формами

Щоб додати ще одну форму в проект, натиснемо на ім'я проекту у вікні Solution Explorer (Оглядач рішень) правою кнопкою миші і виберемо Add (Додати) -> Windows Form ...

Дамо новій формі якусь ім'я, наприклад, Form2.cs
Отже, у нас в проект була додана друга форма. Тепер спробуємо здійснити взаємодія між двома формами. Припустимо, перша форма після натискання на кнопку буде викликати другу форму. По-перше, додамо на першу форму Form1 кнопку і подвійним клацанням по кнопці перейдемо в файл коду. Отже, ми потрапимо в обробник події натискання кнопки, який створюється за замовчуванням після подвійного клацання по кнопці:

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

}
```

Тепер додамо в нього код виклику другий форми. У нас друга форма називається Form2, тому спочатку ми створюємо об'єкт даного класу, а потім для його відображення на екрані викликаємо метод Show:

```
private void button1_Click (object sender, EventArgs e)
{
   Form2 newForm = new Form2 ();
   newForm.Show ();
}
```

Тепер зробимо навпаки - щоб друга форма впливала на першу. Поки друга форма не знає про існування першої. Щоб це виправити, треба другий формі якось передати відомості про першу формі. Для цього скористаємося передачею посилання на форму в конструкторі.
Отже перейдемо до другої формі і перейдемо до її коду - натиснемо правою кнопкою миші на форму і виберемо View Code (Перегляд коду). Поки він порожній і містить тільки конструктор. Оскільки C # підтримує перевизначення методів, то ми можемо створити кілька методів і конструкторів з різними параметрами і в залежності від ситуації викликати один з них. Отже, змінимо файл коду другий форми на наступний:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace HelloApp
{
   public partial class Form2: Form
   {
      public Form2 ()
      {
         InitializeComponent ();
      }

      public Form2 (Form1 f)
      {
         InitializeComponent ();
         f.BackColor = Color.Yellow;
      }
   }
}
```

Фактично ми тільки додали тут новий конструктор public Form2 (Form1 f), в якому ми отримуємо першу форму і встановлюємо її фон в жовтий колір. Тепер перейдемо до коду першої форми, де ми викликали другу форму і змінимо його на наступний:

```
private void button1_Click (object sender, EventArgs e)
{
   Form2 newForm = new Form2 (this);
   newForm.Show ();
}
```

Оскільки в даному випадку ключове слово this являє посилання на поточний об'єкт - об'єкт Form1, то при створенні другої форми вона буде отримувати її (посилання) і через неї керувати першою формою.
Тепер після натискання на кнопку у нас буде створена друга форма, яка відразу змінить колір першої форми.

Ми можемо також створювати об'єкти і поточної форми:

```
private void button1_Click (object sender, EventArgs e)
{
   Form1 newForm1 = new Form1 ();
   newForm1.Show ();

   Form2 newForm2 = new Form2 (newForm1);
   newForm2.Show ();
}
```

При роботі з декількома формами треба враховувати, що одна з них є головною - яка запускається першої в файлі Program.cs. Якщо у нас одночасно відкрита купа форм, то при закритті головної закривається все додаток і разом з ним всі інші форми.

I курс, 2 семестр
Семестровий модуль 1.

ЛЕКЦІЯ 1.
ПОДІЇ В WINDOWS FORMS

Події форми

Для взаємодії з користувачем в Windows Forms використовується механізм подій. Події в Windows Forms представляють стандартні події на C #, тільки що застосовуються до візуальним компонентам і підкоряються тим же правилам, що події в C #. Але створення обробників подій в Windows Forms все ж має деякі особливості.
Перш за все в WinForms є певний стандартний набір подій, який здебільшого є у всіх візуальних компонентів. Окремі елементи додають свої події, але принципи роботи з ними будуть схожі. Щоб подивитися всі події елемента, нам треба вибрати цей елемент в поле графічного дизайнера і перейти до вкладки подій на панелі форм.
Щоб додати оброблювач, можна просто два рази натиснути по порожньому полю поруч з назвою події, і після цього Visual Studio автоматично згенерує обробник події. Наприклад, натиснемо для створення обробника для події Load.

І в цьому полі відобразиться назва методу обробника події Load. За замовчуванням він називається Form1_Load.
Якщо ми перейдемо в файл коду форми Form1.cs, то побачимо автосгенерірованний метод Form1_Load:

```
public partial class Form1: Form
{
   public Form1 ()
   {
      InitializeComponent ();
   }

   private void Form1_Load (object sender, EventArgs e)
   {

   }
}
```

І при кожному завантаженні форми буде спрацьовувати код в обробнику Form1_Load.
Як правило, більшість обробників різних візуальних компонентів мають два параметри: sender - об'єкт, який ініціював подія, і аргумент, який зберігає інформацію про подію (в даному випадку EventArgs e).

Для додавання обробника використовується стандартний синтаксис C #:
this.Load + = new System.EventHandler (this.Form1_Load)
Тому якщо ми захочемо видалити створений подібним чином оброблювач, то нам треба не тільки видалити метод з коду форми в Form1.cs, але і видалити додавання обробника в цьому файлі.
Однак ми можемо додавати обробники подій і програмно, наприклад, в конструкторі форми:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace HelloApp
{
   public partial class Form1: Form
   {
      public Form1 ()
      {
         InitializeComponent ();
         this.Load + = LoadEvent;
      }

      private void Form1_Load (object sender, EventArgs e)
      {
      }

      private void LoadEvent (object sender, EventArgs e)
      {
         this.BackColor = Color.Yellow;
      }
   }
}
```

Крім раніше створеного обробника Form1_Load тут також доданий інший обробник завантаження форми: *this.Load + = LoadEvent;* , який встановлює в якості фону жовтий колір.

Контейнери

Для організації елементів управління в пов'язані групи існують спеціальні елементи - контейнери. Наприклад, Panel, FlowLayoutPanel, SplitContainer, GroupBox. Ту ж форму також можна віднести до контейнерів. Використання контейнерів полегшує управління елементами, а також надає формі певний візуальний стиль.
Всі контейнери мають властивість **Controls**, яке містить всі елементи даного контейнера. Коли ми переносимо який-небудь елемент з панелі інструментів на контейнер, наприклад, кнопку, вона автоматично додається в дану колекцію даного контейнера. Або ми також можемо додати елемент керування динамічно за допомогою коду в цю ж колекцію.

Динамічне додавання елементів

Додамо на форму кнопку динамічно. Для цього додамо подія завантаження форми, в якому буде створюватися новий елемент управління. Це можна зробити або за допомогою коду, або візуальним образом.
За допомогою перетягування елементів з Панелі Інструментів ми можемо легко додати нові елементи на форму. Однак такий спосіб досить обмежений, оскільки дуже часто потрібно

динамічно створювати (видаляти) елементи на формі.
Для динамічного додавання елементів створимо обробник події завантаження форми в файлі коду:

```
private void Form1_Load (object sender, EventArgs e)
{
}
```

Тепер додамо в нього код додавання кнопки на форму:

```
private void Form1_Load (object sender, EventArgs e)
{
   Button helloButton = new Button ();
   helloButton.BackColor = Color.LightGray;
   helloButton.ForeColor = Color.DarkGray;
   helloButton.Location = new Point (10, 10);
   helloButton.Text = "Привіт";
   this.Controls.Add (helloButton);
}
```

Спочатку ми створюємо кнопку і встановлюємо її властивості. Потім, використовуючи метод Controls.Add ми додаємо її в колекцію елементів форми. Якби ми це не зробили, ми б кнопку не побачили, оскільки в цьому випадку для нашої форми її просто не існувало б.
За допомогою методу *Controls.Remove ()* можна видалити раніше доданий елемент з форми:

```
this.Controls.Remove (helloButton);
```

Хоча в даному випадку в якості контейнера використовувалася форма, але при додаванні і видаленні елементів з будь-якого іншого контейнера, наприклад, GroupBox, буде застосовуватися всі ті ж методи.

ЛЕКЦІЯ 2.
ЕЛЕМЕНТИ «КОНТЕЙНЕРИ»

GroupBox є спеціальний контейнер, який обмежений від решти форми кордоном. Він має заголовок, який встановлюється через властивість Text. Щоб зробити GroupBox без заголовка, як значення властивості Text просто встановлюється порожній рядок.
Нерідко цей елемент використовується для групування перемикачів - елементів RadioButton, так як дозволяє розмежувати їх групи.
Елемент **Panel** являє панель і також, як і GroupBox, об'єднує елементи в групи. Вона може візуально зливатися з іншою формою, якщо вона має те ж значення кольору фону в властивості BackColor, що і форма. Щоб її виділити можна крім кольору вказати для елемента кордону за допомогою властивості BorderStyle, яке за замовчуванням має значення None, тобто відсутність кордонів.
Також якщо панель має багато елементів, які виходять за її межі, ми можемо зробити прокручуємо панель, встановивши її властивість AutoScroll в true
Також, як і форма, GroupBox і Panel мають колекції елементів, і ми також можемо динамічно додавати в ці контейнери елементи. Наприклад, на формі є елемент GroupBox, який має ім'я groupBox1:

```
private void Form1_Load (object sender, EventArgs e)
{
   Button helloButton = new Button ();
   helloButton.BackColor = Color.LightGray;
   helloButton.ForeColor = Color.Red;
   helloButton.Location = new Point (30, 30);
   helloButton.Text = "Привіт";
   groupBox1.Controls.Add (helloButton);
}
```

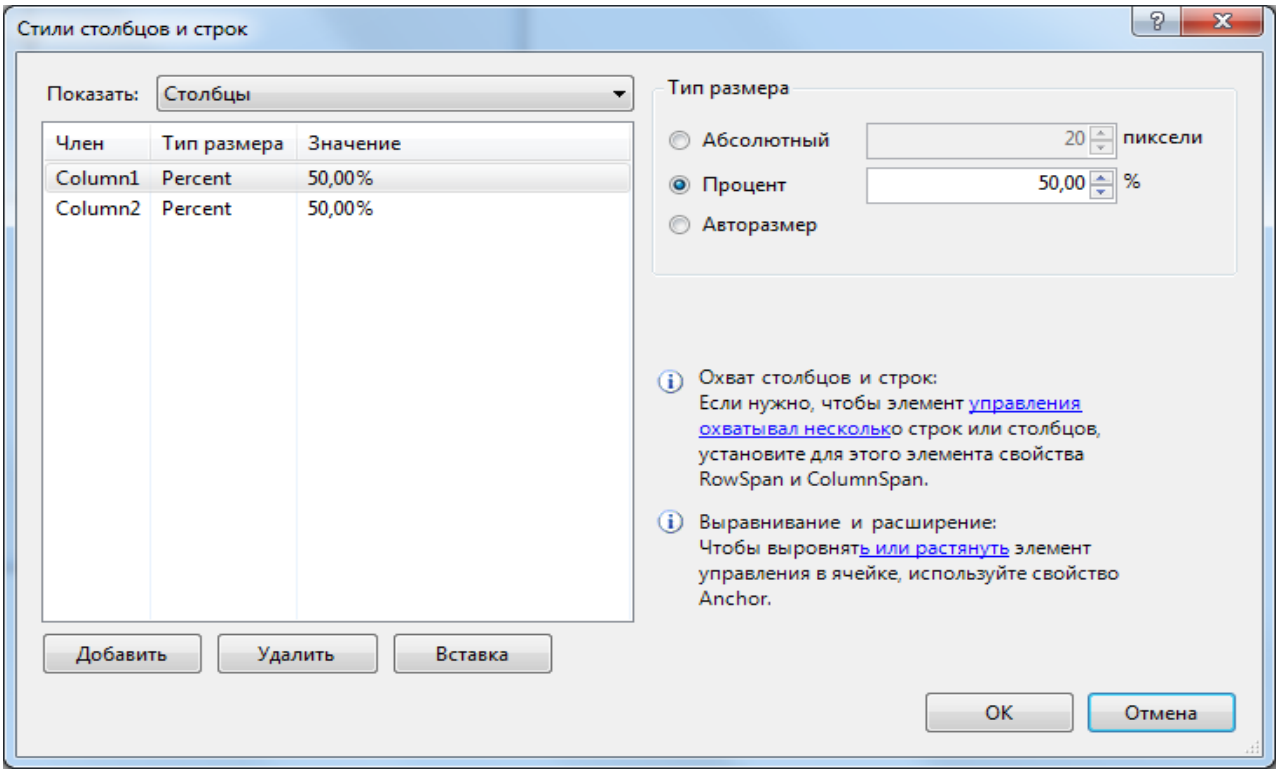
Для вказівки розташування елемента в контейнері ми використовуємо структуру Point*: new Point (30, 30)*; , якої в конструкторі передаємо розміщення по осях X і Y. Ці координати встановлюються щодо лівого верхнього кута контейнера - тобто в даному випадку елемента GroupBox.
При цьому треба враховувати, що контейнером верхнього рівня є форма, а елемент groupBox1 сам знаходиться в колекції елементів форми. І при бажанні ми могли б видалити його:
1 this.Controls.Remove (groupBox1);

FlowLayoutPanel

Елемент FlowLayoutPanel є успадкований від класу Panel, і тому всі його властивості. Однак при цьому додаючи додаткову функціональність. Так, цей елемент дозволяє змінювати позиціонування і компоновку дочірніх елементів при зміні розмірів форми під час виконання програми.
Властивість елемента FlowDirection дозволяє задати напрямок, в якому спрямовані дочірні елементи. За замовчуванням має значення LeftToRight - тобто елементи будуть розташовуватися починаючи від лівого верхнього краю. Наступні елементи будуть йти вправо. Це властивість також може набувати таких значень:
RightToLeft - елементи розташовуються від правого верхнього кута в лівий бік
TopDown - елементи розташовуються від лівого верхнього кута і йдуть вниз
BottomUp - елементи розташовуються від лівого нижнього кута і йдуть вгору
При розташуванні елементів важливу роль відіграє властивість WrapContents. За замовчуванням воно має значення True. Це дозволяє переносити елементи, які не поміщаються в FlowLayoutPanel, на новий рядок або в новий стовпець. Якщо воно має значення False, то елементи не переносяться, а до контейнера просто додаються смуги прокрутки, якщо властивість AutoScroll одно true.

TableLayoutPanel

Елемент TableLayoutPanel також перевизначає панель і має в своєму розпорядженні дочірні елементи управління у вигляді таблиці, де для кожного елемента є своя осередок. Якщо нам хочеться помістити в клітинку більше одного елемента, то в цей осередок додається інший компонент TableLayoutPanel, в який потім вкладаються інші елементи.
Щоб встановити необхідну кількість рядки стовпців таблиці, ми можемо використовувати властивості Rows і Columns відповідно. Вибравши один з цих пунктів у вікні Properties (Властивості), нам відобразиться наступне вікно для настройки стовпців і рядків:



В поле Size Type ми можемо вказати розмір стовпців / рядків. Нам доступні три можливі варіанти:
Absolute: задається абсолютна величина для рядків або стовпців в пікселях
Percent: задається відносний розмір у відсотках. Якщо нам треба створити гумовий дизайн форми, щоб її рядки і стовпці, а також елементи управління в осередках таблиці автоматично масштабувати при зміні розмірів форми, то нам потрібно використовувати саме цю опцію
AutoSize: висота рядків і ширина стовпців задається автоматично в залежності від розміру найбільшої в рядку або стовпці осередки
Також ми можемо комбінувати ці значення, наприклад, один стовпець може бути фіксованим з абсолютною шириною, а решта стовпці можуть мати ширину у відсотках.
У цьому діалоговому вікні ми також можемо додати або видалити рядки і стовпці. У той же час графічний дизайнер в Visual Studio не завжди відразу відображає зміни в таблиці - додавання або видалення рядків і стовпців, зміна їх розмірів, тому, якщо змін на формі ніяких не відбувається, треба її закрити і потім відкрити заново в графічному дизайнера.
Отже, наприклад, у мене є три стовпці і три рядки розмір у яких однаковий - 33.33%. У кожну клітинку таблиці додано кнопку, у якій встановлено властивість Dock = Fill.
Якщо змінити розміри форми, то автоматично масштабуються і рядки і стовпці разом з ув'язненими в них кнопками.
Що досить зручно для створення масштабованих інтерфейсів.
У коді динамічно ми можемо змінювати значення стовпців і рядків. Причому всі стовпці представлені типом ColumnStyle, а рядки - типом RowStyle:

```
tableLayoutPanel1.RowStyles [0] .SizeType = SizeType.Percent;
tableLayoutPanel1.RowStyles [0] .Height = 40;

tableLayoutPanel1.ColumnStyles [0] .SizeType = SizeType.Absolute;
tableLayoutPanel1.ColumnStyles [0] .Width = 50;
```

Щоб установити тривалість в ColumnStyle і RowStyle визначено властивість SizeType, яке приймає одне із значень однойменного перерахування SizeType
Додавання елемента в контейнер TableLayoutPanel має свої особливості. Ми можемо додати його як в наступну вільну комірку або можемо явно вказати елемент таблиці:

```
Button saveButton = new Button ();
// Додаємо кнопку в наступну вільну комірку
tableLayoutPanel1.Controls.Add (saveButton);
// Додаємо кнопку в клітинку (2,2)
tableLayoutPanel1.Controls.Add (saveButton, 2, 2);
```

В даному випадку додаємо кнопку в клітинку, утворену на перетині третього стовпчика і третього рядка. Правда, якщо у нас немає стільки рядків і стовпців, то система автоматично вибере потрібний осередок для додавання.

ЛЕКЦІЯ 3.
ОСНОВИ РОБОТИ З ПЕРЕЛІЧУВАНИМИ ТИПАМИ ТА СТРУКТУРАМИ

Основи роботи з перелічуваними типами

Коли ви створюєте програму, часто буває зручно створити множину символьних імен для базових числових значень.

Розглянемо практичну задачу. Припустимо, що ви пишете програму у якій використовуєте інформацію про дні тижня. До цього ми визначали дні тижня по номеру дня. Наприклад:

Визначення дня тижня за номером.

```
    static void Main(string[] args)
    {
      int a = 4;
      switch (a)
      {
        case 1:
          Console.WriteLine("Понеділок");
          break;
        case 2:
           Console.WriteLine("Вівторок");
           break;
         case 3:
           Console.WriteLine("Середа");
           break;
         case 4:
           Console.WriteLine("Четвер");
           break;
         case 5:
           Console.WriteLine("Пятниця");
           break;
         case 6:
          Console.WriteLine("Субота");
          break;
         case 7:
          Console.WriteLine("Неділя");
          break;
       }
}
```

Проте подібний варіант заставляє програміста постійно пам’ятати, якій цифрі відповідає конкретний день тижня. Краще було б іменувати ці змінні.

У подібних ситуаціях використовуються перелічувані типи.

Перелічуваний тип (enum) - це визначений програмістом тип, який може приймати тільки обмежений набір значень.

Для перелічуваних типів затверджені наступні типи: **byte, sbyte, short, ushort, int, uint, long** або **ulong.**

Загальний вигляд запису перелічуваного типу:

```
модиф_доступу enum назва : тип
{
  Елемент1,
  Елемент2,
  ...,
  ЕлементN,
}
```

Модифікатори доступу можуть бути:
public – доступний для усіх.

private – доступний лише всередині поточного класу.

Перелічуваний тип "Кольори".

```
public enum MyColors
{
  Red,
  Yellow,
  Green
}
```

Перелічуваний тип "Арифметичні операції".

```
public enum Operations
{
  Plus,
  Minus,
  Divide,
  Multiply
}
```

Для днів тижня перелічуваний тип матиме вигляд:

Перелічуваний тип "Дні тижня".

```
enum Days
{
  Mon,   //0
  Tue,   //1
  Wed,   //2
  Thu,   //3
  Fri,   //4
  Sat,   //5
  Sun   //6
}
```

Перелічувальний тип Days визначає чотири іменовані константи, відповідні деяким числовим значенням. У C# перший елемент за замовчуванням має нульовий індекс (0) і далі по наростаючій (n + 1).

Ви можете змінити цю поведінку, як вам потрібно:

Перелічуваний тип "Дні тижня".

```
enum Days
      {
       Mon = 10,
        Tue,       //11
        Wed,       //12
        Thu,       //13
        Fri,       //14
        Sat,       //15
      Sun          //16
      }
```

Номери елементів перелічувального типу не обов'язково мають бути послідовними. Якщо (з будь-якої причини) є сенс визначити Days таким чином, компілятор заперечувати не буде:

Перелічуваний тип "Дні тижня".

```
      enum Days
      {
        Mon = 10,
        Tue,          //11
        Wed = 75,
       Thu = 100,
        Fri,          //101
        Sat,          //102
       Sun            //103
      }
```

Для зберігання кожного елементу перелічувального типу за замовчуванням використовується клас System.Int32. Ви також можете змінити цю поведінку. Наприклад, якщо ви хочете, аби внутрішні елементи Days зберігалися як значення типе byte, а не як int, можна написати наступне:

Перелічуваний тип "Дні тижня".

```
      enum Days : byte
      {
        Mon = 10,
        ...,
        Sat,
        Sun
      }
```

Тоді задача про визначення дня тижня буде реалізована наступним чином:

"Дні тижня".

```
1: class Program
   {
     enum Days
     {
       Mon,
       Tue,
       Wed,
       Thu,
       Fri,
        Sat,
        Sun
     }

     static void Main(string[] args)
     {
       Days day = Days.Thu;
      switch (day)
       {
         case Days.Mon:
           Console.WriteLine("Понеділок");
           break;
         case Days.Tue:
           Console.WriteLine("Вівторок");
           break;
         case Days.Wed:
           Console.WriteLine("Середа");
           break;
         case Days.Thu:
           Console.WriteLine("Четвер");
           break;
         case Days.Fri:
           Console.WriteLine("Пятниця");
           break;
         case Days.Sat:
           Console.WriteLine("Субота");
           break;
         case Days.Sun:
           Console.WriteLine("Неділя");
           break;
       }
     }
   }
```

Даний приклад є лише демонстраційним і створювати такий перелічуваний тип немає жодної потреби, адже існує уже вбудований аналогічний тип DayOfWeek.

Структури: призначення, синтаксис

У C# існує багато різних типів даних для представлення інформації. Проте всі обєкти реального світу описати розробникам усе-таки не вдалося :). Наприклад, якщо потрібно описати поняття студент, квартира, веб-сайт, комп'ютер та ін. є потреба створювати користувацькі типи даних. До таких типів даних належать класи та структури. Класи по суті є розширенням структур, вони будуть розглянуті пізніше.

Структури в C # практично нічим не відрізняються від структур на будь-яких інших мовах. Відмінності спостерігаються лише на більш низькому рівні. В основному це стосується того, що для структур в C # не існує базового класу. Але в той же час структури є похідними від типу ValueType.

Дуже узагальнивши поняття структури можна визначити його, як згруповану сукупність ознак(даних) певного об'єкта, методів управління цими ознаками та доступу до них ©.

Структури визначаються за допомогою ключового слова struct, наприклад:

```
модиф_доступу struct Назва
{
  // Поля, властивості, методи...
}
```

Структури використовують велику частину того ж синтаксису, що і класи, проте вони більш обмежені в порівнянні з ними:

У оголошенні структури поля не можуть ініціалізувати до тих пір, поки вони будуть оголошені як постійні або статичні.

Структура може не оголошувати використовуваний за замовчуванням конструктор (конструктор без параметрів) або деструктор.

Структури копіюються при присвоєнні. При присвоєнні структури у нову змінну виконується копіювання всіх даних, а будь-яка зміна нової копії не впливає на дані у вихідній копії.

Структури є значимими типами, а класи - посилальними типами.
Структури можуть бути створені без використання оператора new.
Структури можуть оголошувати конструктори, що мають параметри.

Структура не може наслідуватися від іншої структури або класу і не може бути основою для інших класів. Всі структури наслідуються безпосередньо від System.ValueType, який наслідується від System.Object.

Структури можуть реалізовувати інтерфейси.

Структура може використовуватися як тип, що допускає значення NULL, і їй можна призначити значення NULL.

Структура - це набір залежних один від одного змінних. Залежність тут виключно логічна і визначається умовами задачі. Аби стало зрозуміло, розглянемо простий приклад. Допустимо, ми пишемо програму, що друкує довідки для студентів. Всі довідки мають один і той же вигляд і текст, окрім наступних полів: імя, прізвище, курс, факультет, дата народження. Це залежні дані і їх можна представити у вигляді структури, наприклад так:

Структура "Студент".

```
struct Student
{
  public string _firstName;
  public string _lastName;
  public DateTime _dateOfBirth;
  public string _faculty;
  public int _course;
}
```

Усі елементи у представленій вище структурі є полями. Модифікатор доступу **public** до полів означає, що доступ до цього поля є повним у всіх блоках програми.

Наша структура називається Student і має 5 полів. Після того, як структура оголошена, ми можемо з нею працювати. Ось невеликий приклад:

Ввести з клавіатури значення масиву можна наступним чином:

Робота зі структурою "Студент".

```
struct Student
{
  public string firstName;
  public string lastName;
  public DateTime dateOfBirth;
  public string faculty;
  public int course;
}

static void Main(string[] args)
{
   Student student = new Student();
   student.firstName = "Дмитро";
   student.lastName = "Попов";
   student.faculty = "гуманітарний";
   student.course = 3;
   student.dateOfBirth = DateTime.Parse("02/05/1990");

   Console.WriteLine("\t\t\tДОВІДКА\n");
   Console.WriteLine(" підтвреджує, що");
Console.WriteLine(" {0} {1} дійсно навчається на {2}-му курсі.", student.firstName,
student.lastName, student.course);
   Console.WriteLine(" Дата народження: " + student.dateOfBirth.ToShortDateString());
   Console.WriteLine(" Факультет: " + student.faculty);
}
```

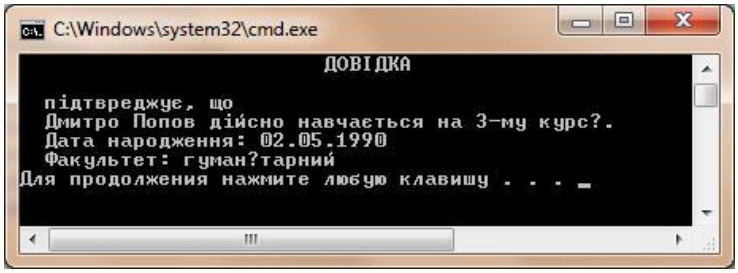


Рис. Результат виконання прикладу.

Структури підримують також і методи. Виведення інформації про студента можна винести у метод структури і викликати у програмі. Для цього створимо метод Print. Також напишемо метод, який буде повертати повне ім'я студента, наприклад, Попов Дмитро.

Робота зі структурою "Студент".

```
struct Student
{
    public string firstName;
    public string lastName;
    public DateTime dateOfBirth;
    public string faculty;
    public int course;

    public void Print()
    {
        Console.WriteLine("\t\t\tДОВІДКА\n");
        Console.WriteLine("  підтвреджує, що");
        Console.WriteLine("  {0} {1} дійсно навчається на {2}-му курсі.", firstName, lastName,
        course);
        Console.WriteLine("  Дата народження: " + dateOfBirth.ToShortDateString());
        Console.WriteLine("  Факультет: " + faculty);
     }

public string GetFullName()
  {
        return firstName + " " + lastName;
     }
}

    static void Main(string[] args)
    {
        Student student = new Student();
        student.firstName = "Дмитро";
        student.lastName = "Попов";
        student.faculty = "гуманітарний";
       student.course = 3;
        student.dateOfBirth = DateTime.Parse("02/05/1990");

      student.Print();
      Console.WriteLine("\n\n" + student.GetFullName() + "\n\n");
}
```

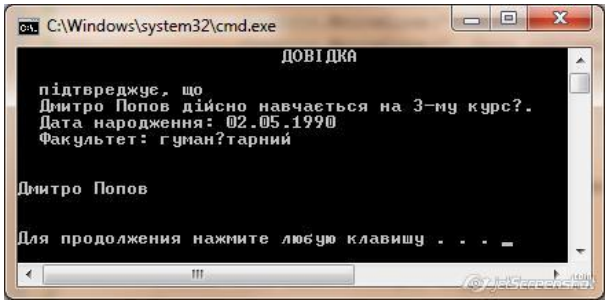


Рис. Результат виконання прикладу.

Структури підтримують конструктори.

У об'єктно-орієнтованому програмуванні конструктор класу (від англ. constructor, інколи скорочують ctor) - спеціальний блок інструкцій, що викликається при створенні об'єкту з використанням ключового слова **new**.

Конструктор схожий з методом, але відрізняється від методу тим, що не має явно оголошеного типу повертаємого значення, не наслідується. Конструктори виділяються наявністю однакового імені з ім'ям класу, в якому оголошуються. Конструкторів може бути одразу кілька.

Робота зі структурою "Студент".

```
struct Student
{
    //Поля
    public string firstName;
    public string lastName;
    public DateTime dateOfBirth;
    public string faculty;
    public int course;

    //Конструктор
    public Student(string _firstName, string _lastName, DateTime _dateOfBirth, string _faculty, int
    _course)
    {
        this.firstName = _firstName;
        this.lastName = _lastName;
        dateOfBirth = _dateOfBirth;
        faculty = _faculty;
        course = _course;
    }

    //Методи
    public void Print()
    {
        Console.WriteLine("\t\t\tДОВІДКА\n");
        Console.WriteLine("  підтвреджує, що");
        Console.WriteLine("  {0} {1} дійсно навчається на {2}-му курсі.", firstName, lastName,
        course);
        Console.WriteLine("  Дата народження: " + dateOfBirth.ToShortDateString());
        Console.WriteLine("  Факультет: " + faculty);
    }

    public string GetFullName()
    {
        return firstName + " " + lastName;
    }  }

static void Main(string[] args)
{
    Student student = new Student("Дмитро", "Попов", DateTime.Parse("02/05/1990"),
    "гуманітарний", 3);
    student.Print();   }
```

Якщо порывняти приклад та 12, то коду приблизно однаково. Але при зростанні кількості обєктів типу Студент буде видно суттєву оптимізацію. Напишемо програму, яка виводить на екран довідки 3-х студентів.

Робота зі структурою "Студент". Масив студентів

```
static void Main(string[] args)
{
    Student[] group = new Student[3];
    group[0] = new Student("Дмитро", "Попов", DateTime.Parse("02/05/1990"), "економічний",
    3);
    group[1] = new Student("Іван", "Петров", DateTime.Parse("02/05/1990"), "економічний", 3);
    group[2] = new Student("Олена", "Чуприна", DateTime.Parse("02/05/1990"), "економічний",
    3);
    foreach (Student student in group)
    {
        student.Print();
    }
}
```

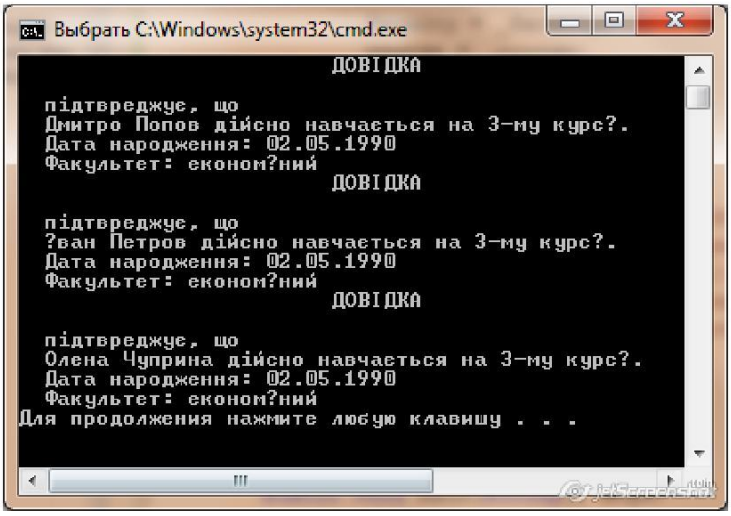


Рис. Результат виконання прикладу.

ЛЕКЦІЯ 4.
Поняття об'єкта та класу. Основні елементи класу

Клас та об'єкт

Об'єктно-орієнтоване програмування і проектування побудоване на класах. Будь-яку програмну систему, побудовану в об'єктному стилі, можна розглядати як сукупність класів, можливо, об'єднаних в проекти, простори імен, рішення, як це робиться при програмуванні у Visual Studio.

Клас - це шаблон, який визначає форму об'єкту. Він задає як дані, так і код, який оперує цими даними.

Об'єкти - це екземпляри класу.
Клас складається із:

1. полів;
2. властивостей;
3. методів;
4. подій;
5. конструкторів;
6. деструкторів;
7. делегатів.

Елементи класу називаються членами класу.
Клас оголошується за допомогою ключового слова class. Синтаксис має наступний вигляд:

Синтаксис оголошення класу.

```
class ім’я_класу
{
  //Оголошення полів
  доступ тип імя_змінної;
  доступ тип імя_змінної;

  //Оголошення методів
  доступ тип_повернення імя_метода(параметри)
  {
   тіло метода;
  }

  доступ тип_повернення імя_метода(параметри)
  {
  тіло метода;
  }
}
```

Розглянемо приклад базового створення класу "Комплексне число".

Клас "Комплексне число".

```
public class ComplexNumber
{
  //Поля
```

```
        private double a;
        private double b;

        //Конструктор
        public ComplexNumber(double a, double b)
        {
           this.a = a;
           this.b = b;
         }

         //Метод
         public override string ToString()
         {
           return a + " " + b + "i";
}
}
```

Доступ до полів, методів та інших членів класу може здійснюватися з різним рівнем доступу:

private доступний лише всередині класу (типу)

protected доступний лише всередині класу та класів-нащадків

internal доступний лише в межах збірки

protected internal доступний лише в межах збірки, лише всередині класу та класів-нащадків

public доступний для усіх

Поля класу

Поля класу синтаксично є звичайними змінними (об'єктами) мови. Їх опис задовольняє звичайним правилам оголошення змінних, про що детально говорилося раніше. Змістовно поля задають представлення тій самій абстракції даних, яку реалізує клас.

Поля характеризують властивості об'єктів класу. Коли створюється новий об'єкт класу, то цей об'єкт є набором полів класу. Два об'єкти одного класу мають один і той же набір полів, але різняться значеннями, що зберігаються в цих полях.

Синтаксис оголошення полів:

модифікатор_доступу тип назва;

Наприклад, оголосимо клас Worker, який має 3 поля: розмір з/п, прізвище, вік.

Клас "Працівник"

```
class Worker
    {
    public int salary;           //Розмір з/п
public string firstname;  //Ім'я
    public string lastname;       //Прізвище
     }
```

Зараз клас працівник нагадує структуру! І це не дивно, адже клас є більш розвиненою структурою.

Методи класу

Змінні(поля) екземплярів і методи - дві основні складові класів. Поки наш клас Worker містить лише дані. Хоча такі класи (без методів) допустимі, більшість класів мають методи.

Методи - це процедури (підпрограми), які маніпулюють даними, визначеними в класі, і у багатьох випадках забезпечують доступ до цих даних. Зазвичай різні частини програми взаємодіють з класом за допомогою його методів. Будь-який метод містить одну або декілька інструкцій.

Кожен метод має ім'я, і саме це ім'я використовується для його виклику. У загальному випадку методу можна привласнити будь-яке ім'я. Але пам'ятаєте, що ім'я Main() зарезервовано для методу,

з якого починається виконання програми. Крім того, як імена методів не можна використовувати ключові слова C#.

Імена методів супроводжуються парою круглих дужок. Наприклад, якщо метод має ім'я getval, то в тексті буде написано getval(). Це допомагає відрізняти імена змінних від імен методів. Формат запису методу такий:

```
доступ тип_повернення імя_метода(параметри)
{
  тіло метода;
}
```

Розширимо наш клас Worker:

Клас "Працівник"

```
class Worker
{
  public int salary;     //Розмір з/п
  public string firstname;  //Ім'я
  public string lastname;   //Прізвище

  //Метод, виводить інформацію про працівника на консоль
  public void DisplayInfo()
  {
     Console.WriteLine("{0} {1}, - {2} грн.", lastname, firstname, salary);
  }
}
```

Зверніть увагу ось на що. Змінні екземпляра salary, lastname і firstname використовуються всередині методу DisplayInfo() без будь-яких атрибутів, тобто їм не передує ні ім'я об'єкту, ні оператор "крапка". Це дуже важливий момент: якщо метод задіює змінну екземпляра, яка визначена в його класі, він робить це безпосередньо, без явного посилання на об'єкт і без оператора "крапка". І це логічно. Адже метод завжди викликається для деякого об'єкту конкретного класу. Таким чином, немає необхідності вказувати усередині методу об'єкт удруге. Це означає, що значення salalry, lastname і firstname всередині методу DisplayInfo() неявно вказують на копії цих змінних, що належать об'єкту, який викликає метод DisplayInfo().

Інші приклади методів:

```
public int GetAge();
protected string GetByName(string name);
protected static bool IsEquals(Class obj1, Class obj2);
```

Повернення значення методом

У загальному випадку існує два варіанти умов для повернення з методу. Перший пов'язаний з виявленням закриваючої фігурної дужки, що позначає кінець тіла методу (як продемонстровано на прикладі методу DisplayInfo()). Другий варіант полягає у виконанні інструкції return. Можливі дві форми використання інструкції return: одна призначена для void-методів (які не повертають значень), а інша - для повернення значень.

Негайне завершення void-методу можна організувати за допомогою наступної форми інструкції return:

```
public void DisplayInfo()
{
  if(salary < 0)
     return;
  Console.WriteLine("{0} {1}, {2}", lastname, firstname, salary);
}
```

Хоча void-методи - не рідкість, більшість методів все ж повертають значення. І справді, здатність повертати значення - одна з найкорисніших якостей методу. Ми вже розглядали приклад повернення значення під час роботи з масивами. Значення, які повертаються методами, використовуються в програмуванні по різному. У одних випадках повернене значення є результатом обчислень, в інших - воно просто означає, успішно чи ні виконана певна операція, а в третіх - воно може бути кодом-стану. Методи повертають викликаючим їх процедурам, використовуючи наступну форму інструкції return:

return значення;

Додамо до нашого класу Працівник ще кілька полів і методів.

Клас "Працівник"

```
class Worker
    {
      public string firstname;  //Ім'я
     public string lastname;   //Прізвище
     public int salary;       //Розмір з/п
      public double bonus;//Бонус до з/п у % від з/п


      //Метод, виводить інформацію про працівника на консоль
       public void DisplayInfo()
       {
         Console.WriteLine("{0} {1}, - {2} грн.", lastname, firstname, salary);
       }
```

```
        //Повертає суму бонусу, яку отримає працівник.
        public double GetBonusSum()
        {
          return bonus*salary;
        }

        //Повертає повну суму, яку отримає працівник.
        public double GetFullSum()
        {
          return salary + GetBonusSum();
        }
      }
```

У цьому прикладі створено поле «бонус» та три додаткових методи, функціональність яких подана у коментарях. Розглянемо приклад програми:

Використання класу "Працівник".

```
    static void Main(string[] args)
    {
      Worker worker1 = new Worker();
      worker1.salary = 250;
      worker1.firstname = "Петро";
      worker1.lastname = "Петров";
      worker1.bonus = 0.12;
      Console.WriteLine("Розмір бонусу: {0}\nВсього з/п: {1}",worker1.GetBonusSum(),
      worker1.GetFullSum());
    }
```

Схожий варіант:
Використання класу "Працівник".

```
    static void Main(string[] args)
    {
      Worker worker1 = new Worker();
      worker1.salary = 250;
      worker1.firstname = "Петро";
      worker1.lastname = "Петров";
      worker1.bonus = 0.12;
      double bonusSum = worker1.GetBonusSum();
      double fullSum = worker1.GetFullSum();
      Console.WriteLine("Розмір бонусу: {0}\nВсього з/п: {1}",bonusSum,fullSum);
}
```

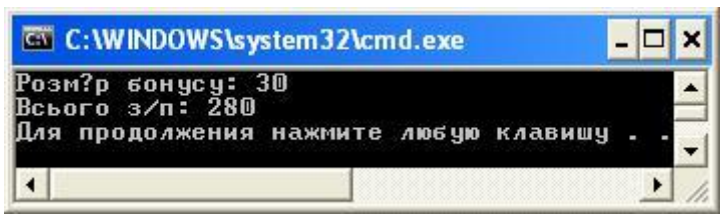


Рис. Результат виконання прикладу.

Як бачимо, результат виконання для обох програм ідентичний. Функції GetBonusSum() та GetFullSum() повертають значення типу double та передають його у першому випадку одразу для виведення на консоль, у другому записують у проміжні змінні. Аналогічні повернення типів даних можна виконувати для усіх типів даних, як базових так і створених користувачем.

Використання параметрів

Під час виклику методу можна передати одне або декілька значень. Значення, яке передається методу, називається аргументом. Змінна всередині методу, яка набуває значення аргументу, називається параметром. Параметри оголошуються всередині круглих дужок, які слідують за ім'ям методу. Синтаксис оголошення параметрів аналогічний синтаксису, вживаному для змінних.

Наприклад, ми можемо визначати суму бонусу, передаючи процент бонусу від з/п у функцію і не записуючи бонус як окреме поле, оскільки є багато працівників, які можуть взагалі не отримати бонус.

```
public double GetBonusSum(double bonusPercent)
{
  return bonusPercent*salary;
}
```

Тоді виклик буде мати вигляд: worker1.GetBonusSum(0.12);

У метод можна передавати безліч аргументів різного типу.

Конструктори

У попередніх прикладах змінні кожного об’єкта встановлювалися "вручну" за допомогою наступної послідовності інструкцій:

```
worker1.salary = 250;
worker1.firstname = "Петро";
worker1.lastname = "Петров";
```

Професіонал ніколи б не використав подібний підхід. І річ не стільки в тому, що таким чином можна просто "забути" про одне або декілька полів, скільки в тому, що існує набагато зручніший спосіб це зробити. Цей спосіб - використання конструктора.

Конструктор ініціалізує об'єкт при його створенні. Він має таке ж ім'я, що і сам клас, а синтаксично подібний до методу. Проте у визначенні конструкторів не вказується тип значення, що повертається. Формат запису конструктора такий:

```
доступ імя_класу()
{
  // тіло конструктора
}
```

Зазвичай конструктор використовується, аби додати змінним екземпляра, визначеним у класі, початкові значення або виконати вихідні дії, необхідні для створення повністю сформованого об'єкту. Крім того, зазвичай як елемент «доступ» використовується модифікатор доступу public, оскільки конструктори, як правило, викликаються поза їх класом.

Всі класи мають конструктори незалежно від того, визначите ви їх чи ні, оскільки C# автоматично надає конструктор за замовчуванням, який ініціалізував всі змінні-члени, що мають типи-значення, нулями, а змінні-члени посилального типу - null-значеннями.

Отже створимо конструктор для класу Worker:

Клас "Працівник".

```
class Worker
      {
        public string firstname;  //Ім'я
        public string
        lastname;                    //Прізвище
        public double salary;   //Розмір з/п
public double bonus;//Бонус до з/п у % від з/п

//Конструктор класу Worker
        public Worker()
         {
           firstname = "empty";
           lastname = "empty";
      salary = 0.0;
           bonus = 0.0;
         }

//Перевантажений конструктор класу Worker
         public Worker(string fname, string lname, double salary, double bonus)
         {
           firstname = fname;
           lastname = lname;
           this.salary = salary;
           this.bonus = bonus;
         }

         //Метод, виводить інформацію про працівника на консоль
         public void DisplayInfo()
         {
Console.WriteLine("{0} {1}, - {2} грн.", lastname, firstname, salary);
         }

         //Повертає суму бонусу, яку отримає працівник.
         public double GetBonusSum()
         {
           return bonus*salary;   }

//Повертає повну суму, яку отримає працівник.
public double GetFullSum()
         {
           return salary + GetBonusSum();
         }
      }
```

Як бачимо, створено два конструктори! Перший (public Worker()) не приймає значень, другий (public Worker(string fname, string lname, double salary, double bonus)) є параметризованим, тобто приймає значення.

Створення двох методів з однаковими іменами, але різними сигнатурами називається перевантаженням методів. Сигнатурою методу є тип повертаємого значення та перелік параметрів. Аналогічно можна перевантажувати конструктори. Детальніше перевантаження методів ми розглянемо під час вивчення наслідування.

Тепер програма маттиме вигляд:

Використання класу "Працівник".

```
static void Main(string[] args)
{
  Worker worker1 = new Worker();

  worker1.DisplayInfo();
  Worker worker2 = new Worker("Степан", "Петров", 25.5,0.5);

  worker2.DisplayInfo();
}
```

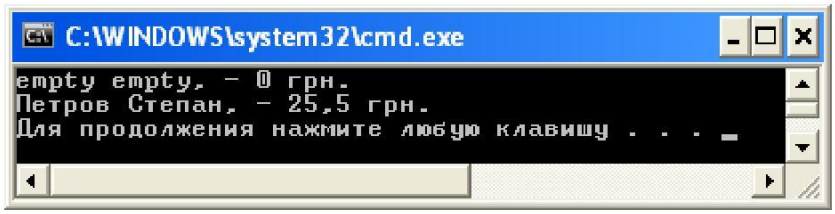


Рис. Результат виконання прикладу.

ЛЕКЦІЯ 5.
ОСНОВИ ООП: ІНКАПСУЛЯЦІЯ

Всі мови ООП, засновані на трьох основоположних концепціях, званих інкапсуляцією, поліморфізмом і спадкуванням.

Інкапсуляція

Інкапсуляція (encapsulation) - це механізм, який об'єднує дані і код, який маніпулює зтім даними, а також захищає і те, і інше від зовнішнього втручання або неправильного використання. В об'єктно-орієнтованому програмуванні код і дані можуть бути об'єднані разом; в цьому випадку говорять, що створюється так званий "чорний ящик". Коли коди і дані об'єднуються таким способом, створюється об'єкт (object). Іншими словами, об'єкт - це те, що підтримує інкапсуляцію.
Усередині об'єкту коди і дані можуть бути закритими (private). Закриті коди або дані доступні тільки для інших частин цього об'єкту. Таким чином, закриті коди і дані недоступні для тих частин програми, які існують поза об'єктом. Якщо коди і дані є відкритими, то, незважаючи на те, що вони задані всередині об'єкта, вони доступні і для інших частин програми. Характерною є ситуація, коли відкрита частина об'єкта використовується для того, щоб забезпечити контрольований інтерфейс закритих елементів об'єкта.
Насправді об'єкт є змінною певного користувачем типу. Може здатися дивним, що об'єкт, який об'єднує коди і дані, можна розглядати як змінну. Однак стосовно об'єктно-орієнтованого програмування це саме так. Кожен елемент даних такого типу є складовою змінної.

Реалізація інкапсуляції традиційними методами доступу і зміни даних

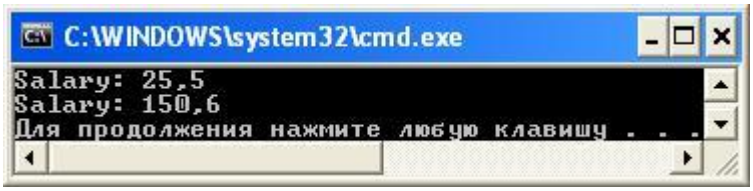
Якщо ви хочете, аби зовнішній світ міг взаємодіяти із закритим полем даних salary, потрібно визначити метод доступу (get) і метод зміни (set). Наприклад:

```
class Worker
 {
    private string firstname;    //Ім'я
    private string lastname;     //Прізвище
    private double salary;       //Розмір з/п
    private double bonus;        //Бонус до з/п у % від з/п
          ...
    public double GetSalary()
    {
      return salary;
    }
    public void SetSalary(double s)
    {
        //здійснити перевірки
      salary = s;
    }
    }
```

Тепер за допомогою методів GetSalary() та SetSalary() ми можемо маніпулювати змінною salary всередині класу. Назвати ваші методи ви можете і по іншому, адже це просто методи, проте бажано робити їх назви відповідно до функцій. Тоді виклик у коді програми матиме наступний вигляд:

```
Worker worker2 = new Worker("Степан", "Петров", 25.5,0.5);
 Console.WriteLine("Salary: {0}",worker2.GetSalary());
 worker2.SetSalary(150.6);
 Console.WriteLine("Salary: {0}", worker2.GetSalary());
```

Результат:



Друга форма інкапсуляції - властивості класу

На противагу традиційним методам доступу і зміни в .NET-мовах інкапсуляцію переважно реалізовують за допомогою *властивостей,* які моделюють відкриті поля даних. Замість того аби заставляти користувача викликати два різні методи для отримання і зміни даних стану, користувач може викликати те, що здається відкритим полем. Для ілюстрації розглянемо властивість Salary, яка замінить два наші методи GetSalary() та SetSalary().

Синтаксис оголошення властивості:

```
доступ тип_повертаємого_ значення назва
{
        get{return змінна(поле);}
        set{змінна(поле) = value;}
}
```

Для нашого класу Worker:

```
  class Worker
  {
    private string firstname;    //Ім'я
    private string lastname;     //Прізвище
    private double salary;       //Розмір з/п
          ...
    public double Salary
    {
      get { return salary; }
      set { salary = value; }
    }  }
```

Властивості лише для читання і лише для запису

При створенні класів ви можете налаштувати властивість доступну лише для запису або лише для читання. Щоб це зробити, просто створіть властивість без відповідного блоку set або get.
Наприклад, властивість тільки для читання:

```
    public double Salary
    {
      get { return salary; }
    }
```

Властивість тільки для запису:

```
    public double Salary
    {
      set { salary = value; }
    }
```

ЛЕКЦІЯ 6.
ОСНОВИ ООП: ПІДТРИМКА НАСЛІДУВАННЯ У C#

Наслідування

Наслідування (inheritance) - це процес, за допомогою якого один об'єкт може набувати властивостей іншого. Точніше, об'єкт може успадковувати основні властивості іншого об'єкта і додавати до них риси, характерні тільки для нього. Наслідування є важливим, оскільки воно дозволяє підтримувати концепцію ієрархії класів (hierarchical classification). Застосування ієрархії класів робить керованими великі потоки інформації. Наприклад, подумайте про опис житлового будинку. Будинок - це частина загального класу, званого будовою. З іншого боку, будова - це частина більш загального класу - конструкції, який є частиною ще більш загального класу об'єктів, який можна назвати створенням рук людини. У кожному разі породжений клас успадковує всі, пов'язані з батьком, якості і додає до них свої власні визначальні характеристики. Без використання ієрархії класів, для кожного об'єкта довелося б задати всі характеристики, які б вичерпні його визначали. Однак при використанні успадкування можна описати об'єкт шляхом визначення того загального класу (або класів), до якого він належить, з тими спеціальними рисами, які роблять об'єкт унікальним. Спадкування грає дуже важливу роль в ООП.

Підтримка наслідування у C#

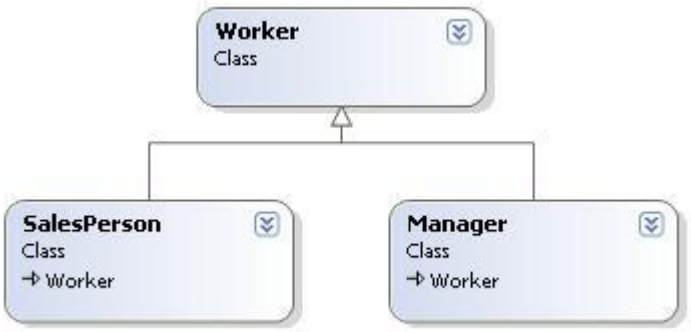
Тепер, коли ми познайомилися з різними способами створення інкапсульованого класу, настав час звернути свою увагу на створення сімейства зв'язаних класів.

Як вже наголошувалося, наслідування - це той аспект ООП, який сприяє багатократному використанню коду.

Наслідування — метод утворення нових класів на основі використання вже існуючих. Наслідування буває двох типів: класичне наслідування (відношення «є») і наслідування відповідно до моделі включения/делегування (відношення «має»).

Давайте почнемо з дослідження класичної моделі «є».

При створенні між класами відношення «є» ви створюєте залежність між типами. Основна ідея класичного наслідування полягає у тому, що нові класи можуть використовувати (і можливо розширювати) функціональність інших класів.



Як показано на малюнку, торгівельний агент «є» співробітником (так само як і менеджер). У класичній моделі наслідування базові класи (такі як Worker) використовуються для визначення загальних характеристик всіх наслідників. Підкласи (такі як SalesPerson і Manager) розширюють цю загальну функціональність, додаючи більш специфічну поведінку.

Для нашого прикладу припустимо, що клас Manager розширює клас Worker, додаючи запис про кількість акцій, якими володіє співробітник, а клас SalesPerson містить обсяги продажів,

здійснені цим агентом. У C# розширення класу виконується за допомогою оператора : (двокрапка) у визначенні класу.

Тоді похідний клас «Торговий агент» матиме наступний вигляд:

```
class SalesPerson : Worker
{
  private double salesQuantity;
  public double SalesQuantity
  {
    get { return salesQuantity; }
    set { salesQuantity = value; }
  }
}
```

Проте, як видно з прикладу, немає конструткора, який би передавав інформацію про агента для класу. Тому розширимо клас і додамо конструтор:

```
class SalesPerson : Worker
  {
    private double salesQuantity;
    public SalesPerson(string fname, string lname, double salary, double bonus, double sQuantity)
      : base(fname,lname,salary,bonus)
    {
      salesQuantity = sQuantity;
    }
    public double SalesQuantity
    {
      get { return salesQuantity; }
      set { salesQuantity = value; }
    }
  }
```

Розберемо код:

1. Наслідування здіснюється за домогою оператора : (двокрапка)
2. Наслідуватися можна одночасно лише від одного класу та багатьох інтерфейсів.
3. Ми не створюємо полів типу firstname, lastname, salary і так далі, всі вони неявно наслідуються від базового класу Worker;
4. У конструктор ми передаємо всю ту ж саму інформацію яку передавали для конструктора Worker + наше нове поле salesQuantity.
5. За допомогою ключовго слова base викликаємо конструктор базовго класу і передаємо йому параметри.
6. Використовуємо властивість для доступу до інформації про обсяги продаж.

Реалізуємо у класі Worker властивості для усіх полів і визначимо метод, який дозволить виводити усю інформацію про працівника на екран:

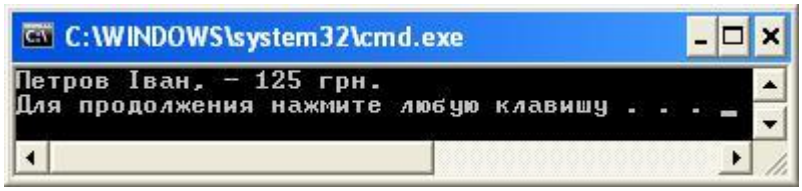
```
public double Salary

    {
      get { return salary; }
      set { salary = value; }
    }
```

```
public string FirstName
{
  get { return firstname;}
  set { firstname = value; }
}
public string LastName
{
  get { return lastname; }
  set { lastname = value; }
}
public double Bonus
{
  get { return bonus; }
  set
  {
    if (bonus >= 0 && bonus < 1)
      bonus = value;
  }
}
```

Тепер напишемо наступну програму:

```
static void Main(string[] args)
    {
      SalesPerson sPerson = new SalesPerson("Петро", "Петров",125, 0.21, 154);
      sPerson.FirstName = "Іван";
      sPerson.;
    }
```

Результат:



У нашому класі SalesPerson немає явно реалізованого методу DisplayInfo() або властивості FirstName, проте ми їх можеми викликати! Тобто ми їх успадкували від батьківського класу Worker.

Майте на увазі, що при наслідуванні інкапсуляція зберігається. Тому похідний клас не може безпосередньо звертатися до закритих членів, визначених в його базовому класі. Тобто не можна наприклад у конструкторі класу SalesPerson записати:

```
public SalesPerson(string fname, string lname, double salary, double bonus, double sQuantity)

    : base(fname,lname,salary,bonus)
{
            //Не можна присвоїти значення.
    firstname = fname;
    salesQuantity = sQuantity;

}
```

Нагадаємо, що усі поля класі Worker приватні (private), тобто закриті від «зовнішнього світу».

ЛЕКЦІЯ 7.
ОСНОВИ ООП: ПІДТРИМКА НАСЛІДУВАННЯ У C#

Ключове слово protected

Як ви вже знаєте, відкриті елементи безпосередньо доступні звідки завгодно, тоді як до закритих елементів не можна дістати доступ з якого-небудь об'єкту, окрім класу, що визначив їх. C# наслідує приклад багатьох інших сучасних об'єктно-орієнтованих мов і надає додатковий рівень доступу - *захищений* (protected) доступ.

Коли базовий клас визначає захищені дані або члени, він створює множину елементів, які можуть бути безпосередньо доступні будь-якому насліднику. Якщо ви хочете дозволити класам Salesperson і Manager безпосередньо звертатися до даних, визначених в класі Worker, початкове визначення класу Worker можна змінити таким чином:

```
class Worker
  {
    protected string firstname;          //Ім'я

    protected string lastname; //Прізвище
    protected double salary;    //Розмір з/п
    protected double bonus;    //Бонус до з/п у % від з/п
        ...
  }
```

Після цього конструкція firstname = fname; стане доступною і програма відкомпілюється.

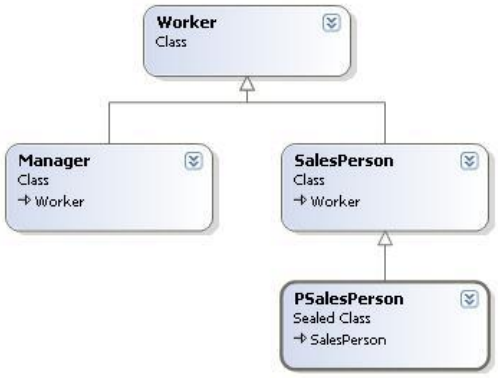
Перевага визначення захищених членів в базовому класі полягає в тому, що похідним класам більше не доведеться діставати доступ до даних за допомогою відкритих методів або властивостей. Вочевидь, є і негатив: коли похідний клас має безпосередній доступ до внутрішніх даних його батьківського класу, існує можливість неумисного обходу бізнес-правил, визначених у відкритих властивостях. При визначенні захищених членів ви створюєте певний рівень довіри між батьківським і дочірнім класом, оскільки компілятор не намагатиметься виявляти які-небудь порушення бізнес-правил. І нарешті, знайте, що з точки зору користувача об'єкту захищені дані вважаються закритими (оскільки користувач знаходиться «поза родинним колом»).

Тому наступний код недопустимий:

```
SalesPerson sPerson = new SalesPerson("Петро", "Петров",125, 0.21, 154);
sPerson.firstname = "Іван";
sPerson.DisplayInfo();
```

Запобігання наслідування - запаковані класи

При створенні відношень базовий клас/підклас можна використовувати поведінку існуючих типів. Проте що якщо ви хочете визначити клас, від якого не можна створювати похідні класи? Наприклад, припустимо, що ви додали ще один клас в простір імен ООР, який розширює існуючий тип SalesPerson. На мал. показана зміна ієрархії.



Класом, що представляє торгівельного агента, що працює за сумісництвом, є PSalesPerson. Припустимо, що нам потрібно гарантувати, що ніхто не зможе створити підклас від PSalesPerson. Аби цей клас не можна було розширювати, в C# використовується ключове слово sealed:

```
sealed class PSalesPerson: SalesPerson
{
  //Поля

  //Властивості
  //Методи
  public PSalesPerson(string fname, string lname, double salary, double bonus, double sQuantity)
    : base(fname, lname, salary, bonus, sQuantity)
  {

        //Інструкції конструктора
  }
}
```

Оскільки клас PSalesPerson запечатано, він не може служити базовим класом для інших типів.

Отже якщо спробувати розширити PSalesPerson, ви отримаєте помилку компіляції:

```
class PPSalesPerson: PSalesPerson      { }
```

Помилка у русифікованому середовищі програмування матиме наступний вигляд:
Ошибка 1 "OOP.PPSalesPerson": нельзя наследовать от запечатанного типа "OOP.PSalesPerson"

Програмування включення/ делегування

Як наголошувалося раніше, наслідування буває двох видів. Тільки що ми розглянули класичне відношення «є». Аби завершити дослідження цього другого стовпа ООП, давайте дослідимо відношення «має» (відоме так само, як модель *включення/делегування).* Припустимо, що ми створили новий клас, що моделює соціальний пакет співробітника:

```
class SocialPackage

{
  //Сума выплаты
  private double socialSum;

  public double SocialSum
```

```
    {

      get { return socialSum; }
      set { socialSum = value; }
    }
  }
```

Вочевидь, було б досить дивним встановлювати відношення «є» між соціальним пакетом (класом SocialPackage) і посадами співробітників. (Менеджер «є» соціальним пакетом? Сумнівно).) Проте повинно бути зрозуміло, що деякий тип відношення між цими двома класами може бути встановлений. Якщо не вдаватися до деталей, можна сказати, що кожен співробітник «має» (has-a) соціальний пакет. Для цього можна додати до полів класу Worker поле SocialPackage таким чином:

```
protected SocialPackage socPackage;
```

Таким чином, ми успішно включили в клас інший об'єкт. Проте для надання функціональності об'єкту, що включається, зовнішньому світу необхідний *делегування.* Делегування - це просто додавання у клас членів, які використовують функціональність об'єкту, що включається. Найпростіший варіант – реалізація властивостей для включеного поля або методів Get, Set.

Вкладені визначення типів

Перш ніж досліджувати останній стовп ООП (поліморфізм), давайте познайомимося з технікою програмування під назвою *вкладені типи.* У C# можна визначити тип (перерахування, клас, інтерфейс, структуру або делегат) безпосередньо в області класу або структури. В цьому випадку вкладений (або «внутрішній») тип вважається *членом* класу, в який він вкладений (тобто «зовнішнього класу»), і з точки зору механізму часу виконання ним можна маніпулювати так само, як будь-яким іншим членом (полем, властивістю, методом, подією і т. д.). Синтаксис, що використовується для створення вкладених типів, досить простий. Розглянемо клас SocialPackage як вкладений.

```
class Worker
  {
    protected string firstname;          //Ім'я
    protected string lastname; //Прізвище
    protected double salary;    //Розмір з/п

    protected double bonus;    //Бонус до з/п у % від з/п
    public class SocialPackage
    {
      //Сума выплаты
      private double socialSum;

      public double SocialSum
      {
        get { return socialSum; }
        set { socialSum = value; }
      }

    }
        . . .
  }
```

Хоча цей синтаксис досить наочний, не завжди зрозуміло, *навіщо* потрібно так робити. Далі представлені аргументи, покликані допомогти в цьому розібратися:

Модель вкладених типів схожа на відношення «має» за винятком того, що у вас є повний контроль над рівнем доступу не до *об'єкту, що* включається, а до внутрішнього *типу.*

Оскільки вкладений тип - це член включеного класу, він може звертатися до закритих членів цього класу.

Частенько вкладений тип корисний лише як допоміжний для зовнішнього класу і не призначений для використання зовнішнім світом.

Коли тип містить інший тип-клас, він може створювати змінні-члени цього типу так само, як і інші елементи даних. Проте, якщо ви хочете використовувати вкладений тип ззовні включаючого типу, тип необхідно кваліфікувати вкладеним типом.

ЛЕКЦІЯ 8.
ОСНОВИ ООП: ПІДТРИМКА ПОЛІМОРФІЗМУ У C#

Поліморфізм

Поліморфізм (polymorphism) (від грецького polymorphos) - це властивість, яка дозволяє одне і те ж ім'я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об'єктно-орієнтованого програмування, є використання одного імені для завдання загальних для класу дій. Виконання кожного конкретного дії буде визначатися типом даних. Наприклад для мови Сі, в якому поліморфізм підтримується недостатньо, знаходження абсолютної величини числа вимагає трьох різних функцій: abs (), labs () і fabs (). Ці функції підраховують і повертають абсолютну величину цілих, довгих цілих і чисел з плаваючою точкою відповідно. В C ++ кожна з цих функцій може бути названа abs (). Тип даних, який використовується при виконанні функції, визначає, яка конкретна версія функції дійсно виконується. В C ++ можна використовувати одне ім'я функції для безлічі різних дій. Це називається перевантаженням функцій (function overloading).
У більш загальному сенсі, концепцією поліморфізму є ідея "один інтерфейс, безліч методів". Це означає, що можна створити загальний інтерфейс для групи близьких за змістом дій. Перевагою поліморфізму є те, що він допомагає мніжать складність програм, дозволяючи використання того ж інтерфейсу для завдання єдиного класу дій. Вибір же конкретного дії, в залежності від ситуації, покладається на компілятор. Вам, як програмісту, не потрібно робити цей вибір самому. Потрібно тільки пам'ятати і використовувати загальний інтерфейс. Приклад з попереднього абзацу показує, як, маючи три імені для функції визначення абсолютної величини числа замість одного, звичайне завдання стає більш складною, ніж це дійсно необхідно.
Поліморфізм може застосовуватися також і до операторів. Фактично в усіх мовах програмування обмежено застосовується поліморфізм, наприклад, в арифметичних операторах. Так, в Сі, символ + використовується для складання цілих, довгих цілих, символьних змінних і чисел з плаваючою точкою. В цьому випадку компілятор автоматично визначає, який тип арифметики потрібно.
В C++ ви можете застосувати цю концепцію і до інших, заданих вами, типам даних. Такий тип поліморфізму називається перевантаженням операторів (operator overloading).
Ключовим в розумінні поліморфізму є те, що він дозволяє вам маніпулювати об'єктами різного ступеня складності шляхом створення спільного для них стандартного інтерфейсу для реалізації схожих дій.

Реалізація поліморфізму у C#

Тепер давайте дослідимо завершальний стовп ООП - *поліморфізм.*
Реалізуємо у класі Worker метод GiveBonus() таким чином:

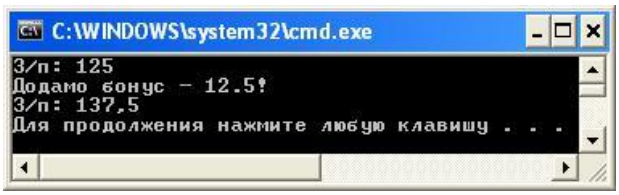
```
public void GiveBonus(float bon)
    {
      salary += bon;     }
```

Оскільки цей метод був визначений як відкритий, ми можемо надавати бонуси як торгівельним агентам, так і менеджерам (а також торгівельним агентам, що працюють за сумісництвом):

```
static void Main(string[] args)
    {

      SalesPerson sPerson = new SalesPerson("Петро",
      "Петров",125, 0.21, 154);
      Console.WriteLine("З/п: {0}", sPerson.Salary);
```

```
        Console.WriteLine("Додамо бонус - 12.5!");
        sPerson.GiveBonus(12.5f);
        Console.WriteLine("З/п: {0}", sPerson.Salary);      }
```

Результат:



Проблема поточного дизайну полягає в тому, що успадкований метод GiveBonus() працює ідентично для всіх підкласів. У ідеалі в бонусі торгівельного представника або торгівельного представника за сумісництвом повинен враховуватися об'єм продажів. Можливо, менеджери повинні отримувати додаткові акції на додаток до збільшення зарплати.

Ключові слова virtual та override

Поліморфізм надає підкласам можливість зміни реалізації методів, визначених у їх базовому класі. Для зміни поточного дизайну необхідно розуміти значення ключових слів virtual та override мови C#. Якщо у базовому класі визначається метод, який *може бути* перекритий підкласом, цей метод має бути віртуальним:

```
class Worker
{   ...
    public virtual void GiveBonus(float bon)
    {
      salary += bon;     }
         ...     }
```

Коли підкласу потрібно перевизначити віртуальний метод, це робиться за допомогою ключового слова override. Наприклад, типи SalesPerson і Manager можуть перекрити метод GiveBonus() таким чином:

```
  class SalesPerson : Worker
  {   ...
      public override void GiveBonus(float bon)
      {
      if (salesQuantity > 100)
        bon += bon * 0.1f;
        salary += bon;     }
          ...     }
```

Зверніть увагу, як кожен перекритий метод може використовувати поведінку за замовчуванням за допомогою ключового слова base. Тобто вам не потрібно повністю повторно реалізовувати логіку методу GiveBonus(), ви можете багато разів задіювати (і, можливо, розширювати) поведінку за замовчуванням батьківського класу.
Наприклад:

```
public override void GiveBonus(float bon)
```

```
    {
 if (salesQuantity >
        100)
bon += bon * 0.1f;
 base.GiveBonus(b
        on);
    }
```

Поняття абстрактного класу

В даний момент базовий клас Worker надає захищені змінні-члени своїм наслідникам, а також підтримує віртуальний метод (GiveBonus ()), який може бути перекритий наслідниками.

Хоча це все і чудово, в поточному дизайні є дивний побічний ефект, що полягає в тому, що ви можете безпосередньо створити екземпляри базового класу Worker:

```
// Що ж це на справді означає?
Worker worker = new Worker();
```

У даному прикладі єдине дійсне призначення базового класу Worker полягає у визначенні загальних полів і членів для всіх підкласів. Зрозуміло, що немає сенсу створювати безпосередній екземпляр цього класу, оскільки тип Worker сам по собі дуже узагальнений. Наприклад, якби я підійшов до вас і сказав:

«Я співробітник!», ваше первше питання було б: «Ну *і що* ти за співробітник?» (консультант, тренер, помічник адміністратора, редактор, співробітник Білого дому і т. д.).

Враховуючи, що багато базових класів є досить туманними сутностями, набагато краще в нашому прикладі було б запобігти можливості безпосереднього створення об'єктів класу Employee в коді. У C# це можна зробити програмно, використовуючи ключове слово abstract:

```
abstract class Worker
{
        Поля
        Методи
        Властивості
}
```

Таким чином створити об’єкт класу не вдасться.

ЛЕКЦІЯ 9.
РОБОТА З ГРАФІКОЮ

Графічний інтерфейс GDI +

Графічний інтерфейс додатків C #, як і інших додатків, призначених для роботи в рамках Microsoft .NET Framework, складається з набору класів. Ці класи інкапсулюють поведінку об'єктів і інструментів, призначених для малювання. Однак перш ніж приступити до опису цих класів, нам необхідно познайомитися з основними поняттями інтерфейсу графічних пристроїв.

Основні поняття

Якщо Ви коли-небудь створювали програми для операційної системи MS-DOS і її аналогів, то пам'ятайте, що такі програми можуть працювати в текстовому або графічному режимі. Використання текстового режиму зазвичай не викликає труднощів. Програми виводять текстові рядки на екран консолі, звертаючись до системних функцій ОС або до функцій базової системи введення-виведення BIOS. Що ж стосується графічного режиму, то його використання приводило до необхідності виконувати в програмах пряме звернення до апаратури відеоадаптера.

Незалежність від апаратури

При створенні ОС Microsoft Windows компанія Microsoft позбавила програмістів від необхідності враховувати апаратні особливості відеоадаптерів, переклавши цю задачу на драйвери відеоадаптерів. Ці драйвери створюються розробниками відеоадаптерів і найкращим чином реалізують можливості апаратури.
Що ж стосується додатків, то для них в складі ОС Microsoft Windows був передбачений набір системних функцій, що реалізують інтерфейс графічних пристроїв (Graphics Device Interface, GDI). Інтерфейс графічних пристроїв GDI, як це можна припустити з назви, призначений для взаємодії додатків Microsoft Windows з графічними пристроями, такими як відеоадаптер, принтер або плоттер.
Коли додатки звертаються до GDI для виконання операції виведення графічного зображення, вони працюють не з реальними (фізичними) пристроями виведення, а з логічними пристроями. Додатки Microsoft Windows не визначають тип відеоадаптера (EGA, VGA, SVGA і т.п.), а працюють з логічним відеоадаптером, що має феноменальні характерристики: здатність відображати практично будь-який колір, у яких велике дозвіл і т.д. Виконуючи запит додатки, GDI звертається до драйверу відповідного пристрою виведення, що працює, в свою чергу, безпосередньо з фізичним пристроєм виведення. В процесі виконання запиту GDI (або драйвер) враховує обмежені можливості відеоадаптера і його апаратні особливості, роблячи необхідні наближення.
Наприклад, додаток може вказати для кольору лінії будь-який з приблизно 16 млн. Квітів, проте не всяке пристрій володіє таким колірним дозволом (обмеження на кількість одночасно відображаються присутні, наприклад, в кишенькових комп'ютерах). Залежно від типу фізичного пристрою, що використовується для виведення, GDI може вибрати для відображення колір, найбільш відповідний запрошенням кольором, і допустимий для пристрою.

Наприклад, якщо пристрій виведення монохромне, замість різних кольорів можуть використовуватися градації сірого кольору. Тому додаток може запитати для виведення будь-який колір, але для малювання буде використаний тільки такий, який може використовувати дане фізичне пристрій.
Така ситуація, коли додаток запитує у ОС Microsoft Windows одне, а отримує інше, виникає не тільки при роботі з кольором. Додаток може запросити для виведення шрифт, описавши його характеристики. Інтерфейс GDI підбере для виведення найбільш підходящий (з його точки зору) шрифт, відповідний опису, і надасть його додатком.

Контекст відображення

З точки зору додатків, інтерфейс GDI складається з контексту відображення та інструментів, призначених для малювання. Контекст відображення можна порівняти з аркушем паперу, на якому додаток малює ту чи іншу графічне зображення, а також пише текст. Інструменти для малювання - це пір'я, кисті (а також шрифти і навіть цілі графічні зображення), за допомогою яких створюється зображення.
Крім контексту відображення та інструментів для малювання, додатків доступні десятки функцій програмного інтерфейсу GDI, призначені для роботи з контекстом відображення і інструментами. Що ж стосується додатків Microsoft .NET Framework, то вони реалізують можливості інтерфейсу GDI + за допомогою набору відповідних класів і інтерфейсів.
У термінах ОС Microsoft Windows контекст відображення (display context) представляє собою структуру даних, що описує пристрій відображення. У цій структурі зберігаються різні характеристики пристрою і набір інструментів для малювання, обраний за замовчуванням. Додаток може вибирати в контекст відображення різні інструменти (наприклад, пір'я різної товщини і кольору, з різними «наконечниками»). Тому якщо Вам треба намалювати лінію червоного або зеленого кольору, перед виконанням операції слід вибрати в контекст відображення відповідне перо.
Зауважимо, що функції малювання GDI, що входять в програмний інтерфейс Win32 API, не мають параметрів, що вказують колір або товщину лінії. Такі параметри зберігаються в контексті відображення.
Додаток може створити контекст відображення не тільки для вікна програми, але і для будь-якого іншого графічного пристрою виведення, наприклад, для принтера. В останньому випадку воно може малювати на принтері різні зображення, використовуючи ті ж функції, що і для малювання у вікні програми.
Можна створити контекст відображення для метафайла. Метафайл - це звичайний файл або файл в пам'яті, в якому зберігаються послідовності команд інтерфейсу GDI. Додаток може виконувати графічний висновок в метафайл як в звичайний пристрій виведення, а потім «програвати» метафайл на реальному пристрої виведення.
Контекст пристрою в термінах ОС Microsoft Windows виступає в ролі сполучної ланки між додатком і драйвером пристрою і являє собою структуру даних розміром приблизно 800 байт. Ця структура даних містить інформацію про те, як потрібно виконувати операції виведення на цьому пристрої (колір і товщину лінії, тип системи координат і т. Д.).
Якщо додаток отримує або створює контекст для пристрою відображення, такий контекст називається контекстом відображення (display context). Тому коли, наприклад, додаток отримує контекст для відображення в одному зі своїх вікон, такий контекст називається контекстом відображення. Якщо ж йому потрібно виконувати операцію виведення для пристрою (для принтера або для екрану дисплея), додаток повинен отримати або створити контекст пристрою (device context).
Слід розуміти, що контексти пристрої та відображення містять описи одних і тих же характеристик і мають однакову структуру. Назва контексту визначається тільки тим, чи стосується контекст до вікна відображення або пристрою виведення.

ЛЕКЦІЯ 10.
РОБОТА З ГРАФІКОЮ

Клас Graphics

Концепція графічного інтерфейсу GDI + дещо відрізняється від концепції «класичного» графічного інтерфейсу GDI, з яким звикли мати справу розробники додатків Microsoft Windows.
Перш за все, це стосується класу Graphics, що реалізує в собі як властивості контексту відображення, так і інструменти, призначені для малювання в цьому контексті.
Для того щоб додаток могло що-небудь намалювати у вікні, воно повинно, перш за все, отримати або створити для цього вікна об'єкт класу Graphics. Далі, користуючись властивостями і методами цього об'єкта, додаток може малювати у вікні різні фігури або текстові рядки.

Подія Paint

Перевіряючи роботу додатка, описаного раніше, Ви напевно помітите одну неприємну особливість - при зміні розмірів вікна частина намальованого зображення або все зображення пропадає. Аналогічна неприємність відбувається і в тому випадку, коли вікно програми перекривається вікном іншої програми.
Проблема в тому, що в додатку не реалізована техніка малювання у вікні, що застосовується у всіх стандартних додатках Microsoft Windows. За своєю логікою спосіб малювання програм Microsoft Windows в корені відрізняється від способу, до якого звикли розробники, що створювали програми для MS-DOS.

Малювання на панелі програм Microsoft Windows

Відомо, що додатки Microsoft Windows не можуть виводити текст або графіку ні за допомогою стандартних функцій бібліотеки компілятора, наприклад таких, як printf, cprintf або putc. Не допоможуть і переривання BIOS, так як додатків Microsoft Windows заборонено їх використовувати (у всякому разі, для виведення на екран). Всі ці функції орієнтовані на консольний висновок в одне-єдине вікно, надане в повне розпорядження програмі MS-DOS.
В ОС Microsoft Windows паралельно працюючі додатки повинні спільно використовувати один загальний екран монітора. Для цього вони створюють перекриваються і переміщувані вікна, в які і виконують висновок тексту або графічних зображень. ОС Microsoft Windows бере на себе всі проблеми, пов'язані з можливим перекриттям або переміщенням вікон, так що правильно спроектовані програми не повинні спеціально піклуватися про відновлення вмісту вікна після його перекриття іншим вікном.
Спосіб, яким додаток Microsoft Windows виводить що-небудь в свої вікна, докорінно відрізняється від способу, використовуваного в програмах MS-DOS.
Програма MS-DOS формує зображення на екрані «розосередженим» чином, тобто в будь-якому місці програми можуть викликатися функції, які виводять щось на екран. Наприклад, відразу після запуску програма може намалювати на екрані діалогову панель, а потім в будь-який момент часу і, що найголовніше, з будь-якого місця програми модифікувати її.

Додатки Microsoft Windows також можуть виводити в створені ними вікна текст або графічні зображення в будь-який момент часу і з будь-якого місця. Саме так чинить наш додаток. Однак зазвичай розробники діють по-іншому.

Повідомлення WM_PAINT

Перш ніж приступити до опису способів малювання в вікнах, застосовуваних додатками .NET Frameworks, розповімо про те, як це роблять «класичні» додатка Microsoft Windows, складені на мовах програмування C або C ++. Це допоможе Вам глибше розібратися в суті проблеми. При необхідності Ви знайдете більш детальну інформацію про це в [4].
ОС Microsoft Windows стежить за переміщенням і зміною розміру вікон і при необхідності сповіщає додатки, про те, що їм слід перемалювати вміст вікна. Для сповіщення в чергу програми записується повідомлення з ідентифікатором WM_PAINT. Отримавши таке повідомлення, функція вікна повинна виконати перерисовку всього вікна або його частини, в залежності від додаткових даних, отриманих разом з повідомленням WM_PAINT. Нагадаємо, що функція вікна виконує обробку всіх повідомлень, що надходять у вікно програми Microsoft Windows.
Для полегшення роботи по відображенню вмісту вікна весь висновок у вікно зазвичай виконують в одному місці додатка - при обробці повідомлення WM_PAINT в функції вікна. Додаток має бути зроблено таким чином, щоб в будь-який момент часу при надходженні повідомлення WM_PAINT функція вікна могла перемальовувати все вікно або будь-яку його частину, задану своїми координатами.
Останнє неважко зробити, якщо додаток буде зберігати де-небудь в пам'яті свій поточний стан, користуючись яким функція вікна зможе перемалювати вікно в будь-який момент часу.
Тут не мається на увазі, що програма має зберігати образ вікна у вигляді графічного зображення і відновлювати його при необхідності, хоча це і можна зробити. Додаток має зберігати інформацію, на підставі якої воно може в будь-який момент часу перемалювати вікно.
Наприклад, якщо додаток виводить на екран дамп оперативної пам'яті, воно повинно зберігати інформацію про початковому адресу відображуваного ділянки пам'яті і розмірі цієї ділянки. При отриманні повідомлення WM_PAINT додаток повинен визначити, яку ділянку вікна необхідно перемалювати і якого діапазону адрес дампа пам'яті цю ділянку відповідає. Потім програма має заново вивести ділянку дампа пам'яті в вікно, опитуючи відповідні адреси і виконуючи перетворення байт пам'яті в символьні, шістнадцяткові або інші використовувані для виведення дампа числа.
Повідомлення WM_PAINT передається функції вікна, якщо стала видна область вікна, прихована раніше іншими вікнами, якщо користувач змінив розмір вікна або виконав операцію прокрутки зображення у вікні. Додаток може передати функції вікна повідомлення WM_PAINT явно, викликаючи функції програмного інтерфейсу Win32 API, такі як UpdateWindow, InvalidateRect або InvalidateRgn.
Іноді ОС Microsoft Windows може сама відновити вміст вікна, не посилаючи повідомлення WM_PAINT. Наприклад, при переміщенні курсору миші або значка згорнутого додатки ОС відновлює вміст вікна. Якщо ж ОС не може відновити вікно, функція вікна отримує від ОС повідомлення WM_PAINT і перемальовує вікно самостійно.
Перед тим як записати повідомлення WM_PAINT в чергу програми, ОС посилає функції вікна повідомлення WM_ERASEBKGND. Якщо функція вікна і не виконує жодних повідомлення WM_ERASEBKGND, передаючи його функції DefWindowProc, остання у відповідь на це повідомлення зафарбовує внутрішню область вікна з використанням кисті, зазначеної в класі вікна (при реєстрації класу вікна).

Тому, якщо функція вікна намалює що-небудь у вікні під час обробки інших повідомлень, відмінних від WM_PAINT, після приходу першого ж повідомлення WM_PAINT намальоване зображення буде зафарбовано. Зауважимо, що зображення, намальоване в вікні програми пропадає саме з цієї причини.
Що ж робити в тому випадку, коли за логікою роботи програми потрібно змінити вміст вікна не під час обробки повідомлення WM_PAINT, а в будь-якому іншому місці програми?
У цьому випадку програма повинна повідомити ОС Microsoft Windows, що необхідно перемалювати частину вікна або все вікно. При цьому в чергу програми буде записано повідомлення WM_PAINT, обробка якого призведе до потрібного результату. Можна вказати ОС, що був змінений прямокутна ділянка вікна або область довільної форми, наприклад еліпс або багатокутник. Для цього призначені функції InvalidateRect і InvalidateRgn програмного інтерфейсу Win32 API.

Зафарбовані фігури

У класі Graphics визначено ряд методів, призначених для малювання зафарбованих фігур.

Методи для малювання зафарбованих фігур

Метод	Опис
FillRectangle	Малювання закрашеного прямокутника
FillRectangles	Малювання безлічі зафарбованих багатокутників
FillPolygon	Малювання закрашеного багатокутника
FillEllipse	Малювання закрашеного еліпса
FillPie	Малювання закрашеного сегмента еліпса
FillClosedCurve	Малювання закрашеного сплайна
FillRegion	Малювання зафарбованою області типу Region

Є дві відмінності методів з префіксом Fill від однойменних методів з префіксом Draw. Перш за все, методи з префіксом Fill малюють зафарбовані фігури, а методи з префіксом Draw - НЕ зафарбовані. Крім цього, в якості першого параметра методам з префіксом Fill передається не перо класу Pen, а кисть класу Brush.

Ще один приклад:

```
SolidBrush redBrush = new SolidBrush (Color.Red);
g.FillEllipse (redBrush, 50, 50, 100, 110);
```

Тут спочатку створюємо кисть червоного кольору як об'єкт класу SolidBrush. Ця кисть потім передається методу FillEllipse в якості першого параметра. Інші параметри методу FillEllipse задають розташування і розміри прямокутника, в який буде вписаний еліпс.

ЛЕКЦІЯ 11.
РОБОТА З ГРАФІКОЮ

Обробка події Paint

Для форм класу System.Windows.Forms передбачений зручний об'єктно-орієнтований спосіб, що дозволяє з додатком при необхідності перемальовувати вікно форми в будь-який момент часу. Коли вся клієнтська область вікна форми або частина цієї області вимагає перемальовування, формі передається подія Paint. Все, що потрібно від програміста, це створити обробник цієї події, наповнивши його необхідною функціональністю.
Для наочної демонстрації методики обробки події Paint підготуємо найпростіше додаток, що малює в своєму вікні текстовий рядок і геометричні фігури.
Перш за все, створюємо проект програми PaintApp, користуючись майстром проектів системи розробки Microsoft Visual Studio .NET. Далі виділіть у вікні дизайнера форму Form1 і відкрийте вкладку подій для цієї форми.
Пошукайте на вкладці рядок події Paint і клацніть її двічі лівою клавішею миші. В результаті буде створений обробник події Form1_Paint (як це видно на рис. 10 6). Цей оброблювач буде отримувати управління щоразу, коли з тих чи інших причин виникне необхідність в перемальовуванні вмісту вікна нашого застосування.

Ось в якому вигляді буде створений обробник події Paint:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
}
```

Оброблювачу Form1_Paint передаються два параметри.
Через перший параметр передається посилання на об'єкт, що викликав подія. У нашому випадку це буде посилання на форму Form1.
Що ж стосується другого параметра, то через нього передається посилання на об'єкт класу PaintEventArgs. Цей об'єкт має дві властивості, доступних тільки для читання - Graphics і ClipRectangle.
Клас Graphics Вам вже знаком - він представляє собою контекст відображення, необхідний для малювання тексту і геометричних фігур. Ми працювали з цим класом в при створенні додатків, розглянутих раніше. Обробник події Paint отримує контекст відображення через свої параметри, тому програмісту не потрібно визначати його спеціальним чином.
Через властивість ClipRectangle передаються кордону області, яку повинен перемалювати обробник події Paint. Ці кордони передаються у вигляді об'єкта класу Rectangle. Властивості цього класу Left, Right, Width і Height, поряд з іншими властивостями, дозволяють визначити розташування і розміри області.
Зауважимо, що в найпростіших випадках обробник події Paint може ігнорувати властивість ClipRectangle, перемальовуючи вміст вікна повністю. Однак процес перемальовування вмісту вікна можна помітно прискорити, якщо перемальовувати не всі вікно, а тільки область, описану властивість ClipRectangle. Прискорення буде особливо помітним, якщо у вікні намальовано багато тексту і геометричних фігур.
Отже, давайте відредагуємо вихідний текст програми PaintApp таким чином, щоб в його вікні була намальована текстовий рядок, прямокутник і еліпс.

Перш за все, створіть в класі Form1 поле text класу string, в якому буде зберігатися текстовий рядок:

```
public string text;
```

Додайте також в конструктор класу Form1 рядок ініціалізації згаданого поля text:

```
public Form1 ()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent ();

  //
  // TODO: Add any constructor code after InitializeComponent call
  //
  text = "Обробка події Paint";
}
```

І, нарешті, Змінити вихідну мову текст обробника події Form1_Paint наступним чином:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Graphics g = e.Graphics;

  g.Clear (Color.White);
  g.DrawString (text, new Font ( "Helvetica", 15),
     Brushes.Black, 0, 0);
  g.DrawRectangle (new Pen (Brushes.Black, 2), 10, 30, 200, 100);
  g.DrawEllipse (new Pen (Brushes.Black, 2), 150, 120, 100, 130);
}
```

Тут в тілі обробника Form1_Paint ми визначили локальну змінну g класу Graphics, призначену для зберігання контексту відображення. Ця змінна ініціалізується за допомогою значення, отриманого з властивості Graphics першого параметра обробника

Form1_Paint:
Graphics g = e.Graphics;
Отримавши контекст відображення, наш обробник події Paint може малювати у відповідному вікні все, що завгодно.
Спочатку ми зафарбовуємо вікно білим кольором, викликаючи для цього метод Clear, визначений у класі Graphics:

```
g.Clear (Color.White);
```

Таким способом ми можемо зафарбувати фон, колір якого заданий для форми у властивості BackColor.

Далі ми викликаємо методи DrawString, DrawRectangle і DrawEllipse, також певні в класі Graphics:

```
g.DrawString (text, new Font ( "Helvetica", 15), Brushes.Black, 0, 0);
g.DrawRectangle (new Pen (Brushes.Black, 2), 10, 30, 200, 100);
g.DrawEllipse (new Pen (Brushes.Black, 2), 150, 120, 100, 130);
```

Перший з них малює текстовий рядок у верхній частині вікна, а два інших - прямокутник і еліпс, відповідно.
Запустивши додаток, Ви зможете переконатися в тому, що намальоване зображення не пропадає при зміні розмірів вікна. Воно заново перемальовується і в тому випадку, якщо вікно програми виявляється тимчасово перекрито вікном іншої програми.

Перемальовування вікон елементів управління

До сих пір ми не робили ніяких спроб безпосереднього малювання в вікнах форм або у вікнах елементів управління. Все, що ми робили, - це розміщення елементів управління у вікні форми і створення обробників подій, що створюються цими елементами управління при маніпуляціях користувача з клавіатурою і мишкою.
Однак самі по собі використані нами раніше елементи управління виконують обробку події Paint, перемальовуючи при необхідності свої вікна. І програмісту немає ніякої необхідності втручатися в цей процес, якщо тільки йому не потрібно наділити той чи інший елемент управління нестандартною поведінкою. Обробка повідомлення Paint прихована всередині класів елементів управління, тому Ви можете створювати досить складні додатки, навіть не знаючи про існування події Paint.
Але як тільки Вам потрібно щось намалювати або написати у вікні програми або у вікні елементу управління, тут не обійтися без обробки події Paint, а також без знань прийомів малювання із застосуванням контексту відображення Graphics.

ЛЕКЦІЯ 12.
Сплайни

Криві Безьє

Сплайн є криву лінію, що сполучає між собою кілька точок.
Крива Безьє, що представляє собою одну з різновидів сплайна, задається мінімум чотирма точками. Дві з них - початкова і кінцева, а дві інші - керуючі. Крива Безьє проходить через початкову та кінцеву точки, а керуючі точки задають вигини кривої лінії.
Для малювання кривих Безьє є два перевантажених набору методів DrawBezier і DrawBeziers:

```
public void DrawBezier (Pen, Point, Point, Point, Point);
public void DrawBezier (Pen, PointF, PointF, PointF, PointF);
public void DrawBezier (Pen, float, float, float, float, float, float,
  float, float);
public void DrawBeziers (Pen, Point []);
public void DrawBeziers (Pen, PointF []);
```

У всіх цих методах перший параметр задає перо, яка буде використана для малювання. Інші параметри задають координати початкової, кінцевої і керуючих точок.
Що стосується методу DrawBeziers, то він дозволяє задавати координати точок у вигляді масивів, що може бути зручно в деяких випадках.

Малюємо криву Безьє в обробнику події Paint:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Pen myPen = new Pen (Color.Black, 2);

  PointF startPt = new PointF (40.0F, 80.0F);
  PointF control1Pt = new PointF (30.0F, 10.0F);
  PointF control2Pt = new PointF (350.0F, 250.0 F);
  PointF endPt = new PointF (400.0F, 100.0F);

  PointF [] myBezierPoints =
  {
    startPt,
    control1Pt,
    control2Pt,
    endPt
  };

  Graphics g = e.Graphics;
  g.Clear (Color.White);
  g.DrawBeziers (myPen, myBezierPoints);
}
```

Тут ми створюємо початкову і кінцеву точки startPt і endPt, через які проходить наша крива, а також керуючі точки control1Pt і control2Pt. Координати всіх точок передаються методу DrawBeziers через масив myBezierPoints. Керуючі точки згинають лінію, як би притягаючи її до себе.

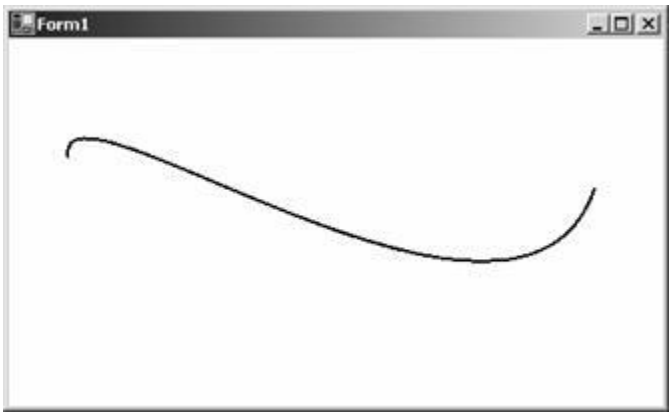


Рис. Малювання кривої Безьє

Канонічні сплайни

На відміну від щойно розглянутих кривих ліній Безьє, лінії канонічного або звичайного сплайна (cardinal spline) проходить через всі задані точки.
Для малювання звичайних сплайнів передбачені методи DrawCurve і DrawClosedCurve. Перший з цих методів малює незамкнуту криву лінію (відкритий сплайн), а другий - замкнуту (закритий сплайн).

У найпростішому випадку методам передається перо і масив з'єднуються точок:
public void DrawCurve (Pen, Point []);
public void DrawCurve (Pen, PointF []);
public void DrawCurveClosed (Pen, Point []);
public void DrawCurveClosed (Pen, PointF []);

Існують версії методів, що дозволяють додатково поставити так звану жорсткість (tension) сплайна. Жорсткість задається у вигляді третього додаткового параметра:

public void DrawCurve (Pen, Point [], float);
public void DrawCurve (Pen, PointF [], float);
public void DrawClosedCurve (Pen, Point [], float, FillMode);
public void DrawClosedCurve (Pen, PointF [], float, FillMode);

За замовчуванням значення жорсткості дорівнює 0,5. При збільшенні цього параметра збільшуються вигини кривої лінії. При жорсткості більшою 1 або меншою 0 крива може перетворитися в петлю.
Методу DrawClosedCurve додатково задається параметр типу FillMode. Якщо значення цього параметра дорівнює FillMode.Alternate, при малюванні самопересекающиеся замкнутих сплайнів будуть чергуватися зафарбовані і не зафарбовані області. Якщо ж значення цього параметра дорівнює FillMode.Winding, більшість замкнутих областей буде зафарбовано.
Малюємо звичайний сплайн, використовуючи точки з тими ж координатами, що і в

попередньому прикладі:

```
using System.Drawing.Drawing2D;
...
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Pen myPen = new Pen (Color.Black, 2);

  PointF pt1 = new PointF (40.0F, 80.0F);
  PointF pt2 = new PointF (30.0F, 10.0F);
  PointF pt3 = new PointF (350.0F, 250.0F);
  PointF pt4 = new PointF (400.0F, 100.0F);

  PointF [] myPoints =
  {
    pt1,
    pt2,
    pt3,
    pt4
  };

  Graphics g = e.Graphics;
  g.Clear (Color.White);
  g.DrawClosedCurve (myPen, myPoints, (float) 0.3,
    FillMode.Alternate);
}
```

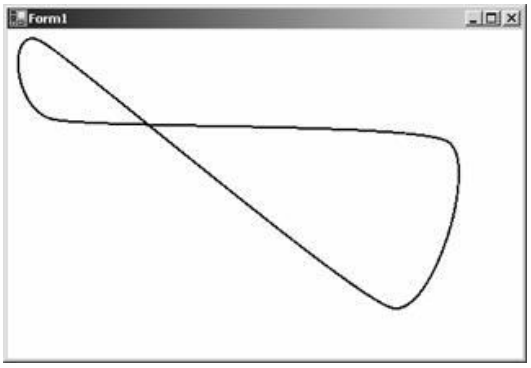


Рис. Малювання канонічного сплайна

Замкнутий сегмент еліпса

Для малювання замкнутого сегмента еліпса (pie) Ви можете скористатися методом DrawPie.

Є 4 перевантажених варіанти цього методу:

```
public void DrawPie (Pen, Rectangle, float, float);
public void DrawPie (Pen, RectangleF, float, float);
public void DrawPie (Pen, int, int, int, int, int, int);
```

public void DrawPie (Pen, float, float, float, float, float, float);

В якості першого параметра методу потрібно передати перо для малювання. Останні два параметри визначають кути, що обмежують сегмент еліпса. Ці кути використовуються таким же чином, як і при малюванні незамкнутого сегмента еліпса методом DrawArc, розглянутим вище. Решта параметра задають розташування і розміри прямокутника, в який вписується сегмент еліпса.

Ось як виглядає обробник події Form1_Paint:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Pen myPen = new Pen (Color.Black, 2);
  int xPos = 10;
  int yPos = 10;
  int width = 250;
  int height = 150;

  int startAngle = 20;
  int endAngle = 75;

  Graphics g = e.Graphics;
  g.Clear (Color.White);

  g.DrawPie (myPen, xPos, yPos, width, height,
    startAngle, endAngle);

  g.DrawRectangle (new Pen (Color.Black, 1), xPos, yPos,
    width, height);
}
```

Тут через змінні xPos, yPos, width і height передаються координати лівого верхнього кута і розміри прямокутної області, в яку вписується сегмент еліпса. Змінні startAngle і endAngle задають кути, що обмежують сегмент.
Для більшої наочності ми малюємо не тільки сегмент еліпса, а й прямокутник, в який цей сегмент вписаний. Для малювання прямокутника ми використовуємо лінію товщиною 1 піксель, а для малювання сегмента - два пікселя.

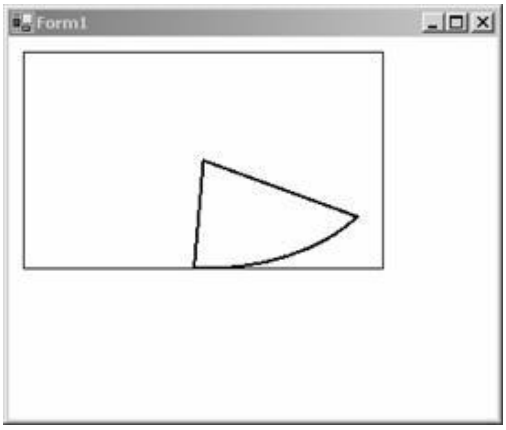


Рис. Малювання сегмента еліпса

ЛЕКЦІЯ 13.
Малювання тексту

Ще один важливий метод, визначений в класі Graphics, це метод DrawString. За допомогою цього методу додатки можуть малювати в своїх вікнах текстові рядки. Нагадаємо, що раніше ми вже користувалися методом DrawString в додатку PaintApp, описаному в розділі «Подія Paint» цієї глави. Ми викликали цей метод в обробнику події Form1_Paint, як це показано нижче:

```
public string text;
...
text = "Обробка події Paint";
...

private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Graphics g = e.Graphics;

  g.Clear (Color.White);
  g.DrawString (text, new Font ( "Helvetica", 15),
    Brushes.Black, 0, 0);
  ...
}
```

В якості першого параметра методу DrawString передається текстовий рядок, який потрібно намалювати.
Другий параметр завдає шрифт. За допомогою третього параметра завдається кисть, із застосуванням якої буде намальований текст. І, нарешті, два останніх параметра визначають координати точки, в якій почнеться малювання тексту.
У класі Graphics визначено кілька перевантажених варіантів методу DrawString:

```
public void DrawString (string, Font, Brush, PointF);
public void DrawString (string, Font, Brush, RectangleF);
public void DrawString (string, Font, Brush, PointF, StringFormat);
public void DrawString (string, Font, Brush, RectangleF,
  StringFormat);
public void DrawString (string, Font, Brush, float, float);
public void DrawString (string, Font, Brush, float, float,
  StringFormat);
```

Параметр типу PointF задає розташування точки виведення тексту. В останніх двох варіантах методу DrawString розташування цієї точки задається за допомогою пари чисел формату float.
Якщо заданий параметр типу RectangleF, то текст буде намальований всередині області, розміри і розташування якої задані цим параметром. У тому випадку, якщо текст вийде за межі області, то він буде обрізаний.
І, нарешті, параметр типу StringFormat дозволяє виконати форматування тексту. Опис цієї можливості Ви знайдете в документації.

Шрифти

Для того щоб малювати текст, використовуються шрифти. ОС Microsoft Windows може працювати з растровими, векторними і масштабованими шрифтами. Крім цього, додатки Microsoft Windows можуть використовувати шрифти, вбудовані в пристрій виведення (зазвичай це принтерні шрифти).

Класифікація шрифтів

Растрові шрифти містять образи всіх символів у вигляді растрових зображень. При цьому для кожного розміру шрифту необхідно мати свій набір символів. Крім того, різні пристрої виведення мають різне співвідношення горизонтальних і вертикальних розмірів пікселя, що призводить до необхідності зберігати окремі набори образів символів не тільки для різних розмірів шрифту, але і для різного співвідношення розмірів пікселя фізичного пристрою відображення.
Растрові шрифти погано піддаються масштабування, так як при цьому похилі лінії контуру символу беруть зазубрений вигляд.
Векторні шрифти зберігаються у вигляді набору векторів, що описують окремі сегменти і лінії контуру символу, тому вони легко масштабуються. Однак їх зовнішній вигляд далекий від ідеального. Як правило, векторні шрифти використовуються для виведення тексту на векторні пристрої, такі, як плоттер.
До складу ОС Microsoft Windows входить не дуже велика кількість шрифтів, однак при необхідності Ви можете придбати додаткові шрифти як окремо, так і в складі різного програмного забезпечення. Наприклад, разом з графічним редактором Corel Draw поставляються сотні різних шрифтів.
Крім звичайних шрифтів існують символьні або декоративні шрифти, які містять замість букв різні значки.

ЛЕКЦІЯ 14.
Шрифти TrueType

Масштабуються шрифти TrueType вперше з'явилися в Microsoft Windows версії 3.1 і сильно вплинули на зростання популярності цієї ОС. Шрифти TrueType піддаються масштабування без істотних спотворень зовнішнього вигляду.
Рис. ілюструє погіршення зовнішнього вигляду реєстрового і векторного шрифтів при збільшеному розмірі символів. Зовнішній вигляд, що масштабується шрифту не погіршився.
Масштабуються шрифти TrueType не тільки зберігають своє накреслення при довільній зміні висоти букв, але і володіють іншими достоїнствами.
Відзначимо, наприклад, можливість виведення рядків тексту, розташованих під будь-яким кутом відносно горизонтальної осі. Растрові і векторні шрифти дозволяють розташовувати рядки тексту тільки в горизонтальному напрямку, що може створити певні труднощі, наприклад, при необхідності надписати назва вулиці на карті міста.
Ще одна перевага шрифтів TrueType пов'язано з тим, що Ви можете вбудувати такий шрифт безпосередньо в документ.
Стандартний набір шрифтів TrueType, що поставляються в складі ОС Microsoft Windows, не завжди задовольняє користувачів. Тому вони набувають додаткові шрифти у незалежних розробників. Однак використання нестандартних шрифтів може привести до проблем при необхідності перенесення документа з одного комп'ютера в інші, так як там потрібного шрифту може не виявитися. Ви, звичайно, можете просто скопіювати потрібний шрифт і перенести його разом з документом, проте така процедура може бути заборонена за умовою ліцензійної угоди з розробниками шрифту.

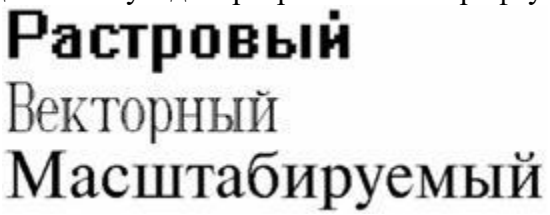


Рис. Растровий, векторний і масштабований шрифти

Проблему перенесення документа на інший комп'ютер зі збереженням прав розробників шрифту можна вирішити, використовуючи шрифти, вбудовані в документ. Користувач може, наприклад, підготувати документ в текстовому процесорі Microsoft Word і вбудувати в нього всі використані шрифти. При перенесенні такого документа на інший комп'ютер ці шрифти можна буде використовувати для перегляду і, можливо, редагування цього (і тільки цього) документа. Можливість редагування з використанням вбудованого шрифту визначається розробником шрифту.
Шрифт TrueType складається із зображень (малюнків) окремих символів - гліфів (glyph). Для внутрішнього уявлення гліфа в файлі шрифту TrueType використовуються опису контурів, причому один гліф може містити кілька таких контурів.

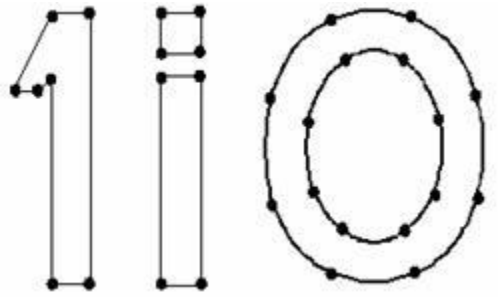


Рис. малюнки символів

Гліфи можуть мати різний зовнішній вигляд (typeface). ОС Microsoft Windows класифікує шрифти за типами, або родин (font family). Ці типи називаються Modern, Roman, Swiss, Script, Decorative.

Додатки Microsoft Windows можуть замовляти шрифт, посилаючись на назву відповідного сімейства, однак в залежності від складу наявних шрифтів ОС Microsoft Windows може дозволити програмі не той шрифт, який би Вам хотілося.
Інша важлива характеристика шрифту - це розмір символів. Для опису вертикального розміру символів шрифту використовуються кілька параметрів.

Рис. Параметри вертикального розміру шрифту

Відлік всіх розмірів виконується від так званої базової лінії (base line) шрифту. Для розмірів використовуються логічні одиниці, які залежать від режиму відображення, встановленого в контексті пристрою.
Загальна висота символів відзначена як Height. Ця висота складається з двох компонентів - Ascent і Descent. Компонент Ascent є висоту від базової лінії з урахуванням таких елементів, як тильда в букві «Ї». Компонент Descent визначає простір, займане символами нижче базової лінії. Сума Ascent і Descent в точності дорівнює Height.
Величина InternalLeading визначає розмір виступаючих елементів символів і може бути дорівнює нулю.
Величина ExternalLeading визначає мінімальний міжрядковий інтервал, рекомендований розробником шрифту. Ваша програма може ігнорувати міжрядковий інтервал, однак в цьому випадку рядки будуть стикатися один з одним, що не завжди прийнятно.
Як бачите, з розмірами символів тут далеко не все так просто, як хотілося б!
Растрові шрифти, які відносяться до одного сімейства, але мають різні розміри букв, зберігаються в окремих файлах. У той же час завдяки можливості масштабування шрифтів

TrueType для них немає необхідності в окремому зберіганні гліфів різних розмірів.
Графічний інтерфейс GDI може виконувати масштабування растрових шрифтів, збільшуючи (але не зменшуючи) розмір букв. Результат такого масштабу при великому розмірі букв зазвичай незадовільний, так як на похилих лініях контуру букв утворюються щербини. Що ж стосується GDI +, то він працює тільки з масштабованими шрифтами, до яких відносяться шрифти TrueType.
Векторні шрифти легко піддаються масштабуванню, тому для зберігання шрифту одного сімейства, але різного розміру, можна використовувати один файл.
Графічний інтерфейс GDI отримує жирне і похиле накреслення растрових шрифтів з нормального за допомогою відповідних алгоритмів потовщення і нахилу шрифту. Такі алгоритми можуть бути використані і для шрифтів TrueType, однак кращих результатів можна досягти при використанні окремих файлів шрифтів TrueType для нормального, жирного і похилого накреслення.
Ще один часто використовуваний атрибут оформлення рядків тексту - підкреслення. Іноді використовується шрифт з перекресленими буквами. GDI виконує підкреслення самостійно, файли шрифтів не містять гліфи з підкресленням.
шрифти OpenType

Вибір шрифту

Перш ніж намалювати текстовий рядок, додаток повинен вибрати шрифт, створивши об'єкт класу Font. У додатку ми вибирали шрифт для малювання тексту методом DrawString:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Graphics g = e.Graphics;

  g.Clear (Color.White);
  g.DrawString (text, new Font ( "Helvetica", 15),
    Brushes.Black, 0, 0);
  ...
}
```

Крім шрифту, методу DrawString необхідно передати кисть для малювання тексту, а також координати точки, в якій цей текст повинен бути намальований. Існують і інші перевантажені варіанти методу DrawString, причому для кожного з них необхідно вказати шрифт.

ЛЕКЦІЯ 15.
Конструктори класу Font

У класі Font існує досить багато конструкторів, за допомогою яких можна підібрати будь-який потрібний Вам шрифт:

public Font (string, float);
public Font (FontFamily, float);
public Font (FontFamily, float, FontStyle);
public Font (FontFamily, float, GraphicsUnit);
public Font (string, float, FontStyle);
public Font (string, float, GraphicsUnit);
public Font (FontFamily, float, FontStyle, GraphicsUnit);
public Font (string, float, FontStyle, GraphicsUnit);
public Font (FontFamily, float, FontStyle, GraphicsUnit, byte);
public Font (string, float, FontStyle, GraphicsUnit, byte);
public Font (FontFamily, float, FontStyle, GraphicsUnit, byte, bool);
public Font (string, float, FontStyle, GraphicsUnit, byte, bool);
public Font (Font, FontStyle);

Першому конструктору потрібно передати назву шрифту (точніше кажучи, назва гарнітури шрифту), а також висоту символів у пунктах (в одному дюймі міститься 72 пункти):

public Font (string, float);

Вибираючи назву гарнітури шрифту, враховуйте, що можна вказувати тільки шрифти TrueType і OpenType. Якщо вказаний Вами шрифт не встановлений на комп'ютері користувача, ОС Microsoft Windows замінить його іншим шрифтом, найбільш підходящим з її «точки зору». Найкраще, якщо програма дозволить користувачеві вибирати шрифт для відображення тексту з числа встановлених в системі шрифтів, тоді з відображенням тексту буде менше проблем.
Останній з конструкторів дозволяє створити шрифт на основі іншого шрифту, змінивши його стиль FontStyle.
Конструктори, у яких є параметр типу byte, дозволяють задавати номер набору символів в термінах GDI.
І, нарешті, останній параметр типу bool дозволяє створювати шрифти з вертикальним розташуванням рядків символів.

Тип шрифту FontStyle

Константи перерахування FontStyle:

Regular, Bold, Italic, Underline, Strikeout

Одиниці виміру розміру шрифту

Параметр конструктора Font типу GraphicsUnit дає Вам можливість вказувати розміри шрифту не тільки в пунктах, а й в інших одиницях виміру.

Константи перерахування GraphicsUnit

Константа	Опис одиниці вимірювання
Display	1/75 частина дюйма
Document	1/300 частина дюйма
Inch	дюйм
Millimeter	міліметр
Pixel	піксель
Point	Пункт (1/72 дюйма)
World	Одиниці глобальних координат (world unit)

Сімейство шрифту FontFamily

За допомогою конструкторів класу Font, що приймають посилання на об'єкт класу FontFamily, можна вибрати шрифт з групи шрифтів, що має схожий дизайн і лише трохи відрізняються в стилі.
Ось конструктори класу FontFamily:

```
public FontFamily (string);
public FontFamily (string, FontCollection);
```

Перший конструктор дозволяє задати ім'я сімейства, а другий - вибрати шрифт з числа шрифтів, встановлених додатком.

Для демонстрації способів створення шрифтів класу Font з використанням конструкторів різного типу створимо додаток.
Ось вихідний текст обробника подій Form1_Paint цього додатка:

```
private void Form1_Paint (object sender,
  System.Windows.Forms.PaintEventArgs e)
{
  Graphics g = e.Graphics;
  g.Clear (Color.White);

  Font f1 = new Font ( "Helvetica", 10);
  g.DrawString ( "Шрифт Helvetica", f1, Brushes.Black, 10, 10);

  Font f2 = new Font (new FontFamily ( "Courier"), 10);

  g.DrawString ( "Шрифт сімейства Courier", f2, Brushes.Black,
    10, 30);

  Font f3 = new Font ( "Times New Roman", 10,
    FontStyle.Bold | FontStyle.Strikeout);

  g.DrawString ( "Шрифт Times New Roman, жирний, перекреслений",
    f3, Brushes.Black, 10, 50);

  Font f4 = new Font ( "Helvetica", 10, GraphicsUnit.Millimeter);
```

```
  g.DrawString ( "Шрифт Helvetica (10 мм)", f4, Brushes.Black,
    10, 70);
}
```

Спочатку створюємо шрифт, вказуючи його ім'я і розмір:

```
Font f1 = new Font ( "Helvetica", 10);
```

Цим конструктором ми вже користувалися раніше в нашій книзі.
Наступний конструктор вибирає не який-небудь конкретний шрифт, а будь-який шрифт сімейства Courier:

```
Font f2 = new Font (new FontFamily ( "Courier"), 10);
```

Такі шрифти є Моноширинний, тобто ширина всіх символів шрифту однакова.
Моноширинних шрифти зазвичай використовуються для оформлення програмних лістингів.
При створенні шрифту можна задати його стиль, як ми це зробили в наступному прикладі:

```
Font f3 = new Font ( "Times New Roman", 10,
  FontStyle.Bold | FontStyle.Strikeout);
```

Тут створюється жирний перекреслений шрифт.
Останній конструктор створює для нас шрифт, вказуючи його висоту не в пунктах, а в міліметрах:

```
Font f4 = new Font ( "Helvetica", 10, GraphicsUnit.Millimeter);
```

СПИСОК ЛІТЕРАТУРИ

Основна література

1. Холзнер, С. *Visual C++6 : учебный курс* .- СПб. : Питер, 2001.- 576 с. : ил..- *1 пр.*
2. Использование C# / за ред. С.Н. Тригуб; пер. с англ. В.С. Иващенко .- Специальное издание.- М.; СПб.; К. : Вильямс, 2002.- 528 с. : ил..- Парал. тит. англ. – *1 пр.*
3. Жоль К.К. *Вступ до сучасної логіки : Навч. посібник для вузів* .- К. : Либідь, 2002.- 152 с. – 10 пр.
4. https://msdn.microsoft.com/ru-ru/library/bb383962(v=vs.90).aspx – Інтерактивний посібник по Visual C#. – *Електронний варіант*

Додаткова література

1. Разработка обслуживаемых программ на языке C#: Виссер Дж. — ДМК Пресс. — 192 с.
2. Microsoft Visual C#. Подробное руководство: Джон Шарп. — Питер. — 848 с.
3. Искусство автономного тестирования с примерами на C#: Ошероув Рой, ДМК Пресс, 2014. — 360 с.
4. C# 4.0: Полное руководство. Генри Шилдт. — 2015. — 1056 с.