МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ
КАФЕДРА КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ

КОНСПЕКТ ЛЕКЦІЙ

з освітнього компонента (навчальної дисципліни)
Об'єктно-орієнтоване програмування
для здобувачів першого (бакалаврського) рівня вищої освіти
за спеціальністю
125 Кібербезпека та захист інформації
(частина 2)

Одеса – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ
КАФЕДРА КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ

КОНСПЕКТ ЛЕКЦІЙ

з освітнього компонента (навчальної дисципліни)
Об'єктно-орієнтоване програмування
для здобувачів першого (бакалаврського) рівня вищої освіти
за спеціальністю
125 Кібербезпека та захист інформації
(частина 2)

Затверджено
на засіданні кафедри кібербезпеки та захисту інформації
Протокол № 1 від 01.09.2025р.

Одеса – 2025

Конспект лекцій з освітнього компонента (навчальної дисципліни) Об’єктно-орієнтоване програмування для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 125 Кібербезпека та захист інформації (частина 2) / Укл.: К.О. Трифонова. – Одеса, 2025. – 36 с.

Укладач: Трифонова К.О., ст. викл.

ЗМІСТ

ВСТУП

Навчальна дисципліна Об’єктно-орієнтоване програмування є необхідною дисципліною для підвищення рівня теоретичних і прикладних знань, що формують фахівця в галузі інформаційних технологій, які сприяють утворенню у здобувачів поглиблених вмінь та навичок розробки програмного забезпечення із застосуванням сучасних парадигм програмування, тобто підготовці спеціалістів для створення, впровадження та підтримки професійно-орієнтованих комп’ютерних технологій у професійній діяльності.

Дисципліна Об’єктно-орієнтоване програмування відповідає освітньо-професійній програмі, навчальному плану підготовки фахівців першого (бакалаврського) рівня вищої освіти спеціальності 125 Кібербезпека та захист інформації і є складовою циклу дисциплін професійної підготовки обов’язкової частини навчального плану.

Дисципліну викладають впродовж третього семестру першого (бакалаврського) рівня. У процесі навчання передбачено лекції, практичні заняття.

Згідно навчального плану передбачено підсумковий контроль у вигляді екзамену.

Робочу програму навчальної дисципліни укладено згідно з вимогами кредитно-модульної системи організації навчального процесу. Програма визначає обсяги компетентностей, які повинен опанувати здобувач відповідно до освітньо-професійної програми, алгоритму вивчення навчального матеріалу дисципліни Об’єктно-орієнтоване програмування, необхідне методичне забезпечення, складові та технологію оцінювання навчальних досягнень.

Предмет навчальної дисципліни Об’єктно-орієнтоване програмування – сучасні технології розробки програмного забезпечення.

Мета вивчення навчальної дисципліни Об’єктно-орієнтоване програмування – забезпечити формування поглибленої системи теоретичних і практичних знань у галузі розробки програмного забезпечення для реалізації здатності використання інформаційно-комунікаційних технологій, сучасних методів і моделей інформаційної безпеки та/або кібербезпеки.

Завдання вивчення дисципліни:

– формувати теоретичні основи та практичні навички застосування методів вилучення, аналізу, обробки та синтезу інформації для формування технічного завдання при розв’язанні задач в галузі інформаційних технологій;

– формувати теоретичні основи та практичні навички сучасних парадигм програмування та способів їх вибору з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі інформаційних технологій;

– поглибити практичні навички кодування програмного забезпечення для розв’язання задач в галузі інформаційних технологій;

– поглибити практичні навички налагодження програмного забезпечення для розв’язання задач в галузі інформаційних технологій.

Стратегічні цілі дисципліни – націлити майбутніх фахівців на застосування отриманих знань у подальшій професійній підготовці та їх наступній практичній діяльності.

ЛЕКЦІЯ №9. ІНТЕРФЕЙСИ, СТРУКТУРИ І ПЕРЕРАХУВАННЯ

У цьому розділі розглядається один з найбільш важливих в C# засобів: інтерфейс, який визначає ряд методів для реалізації в класі. Але оскільки в самому інтерфейсі жоден з методів не реалізується, інтерфейс являє собою чисто логічну конструкцію, що описує функціональні можливості без конкретної їх реалізації.

Крім того, в цьому розділі представлені ще два типи даних C#: структури і перерахування. Структури подібні класам, за винятком того, що вони трактуються як типи значень, а не посилальні типи. А перерахування представляють собою переліки цілочисельних констант. Структури і перерахування розширюють багатий арсенал засобів програмування на C#.

1. Інтерфейси

Іноді в об'єктно-орієнтованому програмуванні корисно визначити, що саме повинен робити клас, але не як він повинен це робити. Прикладом тому може служити згадуваний раніше абстрактний метод. В абстрактному методі визначаються тип, що повертається, і сигнатура методу, але не надається його реалізація. А в похідному класі повинна бути забезпечена своя власна реалізація кожного абстрактного методу, визначеного в його базовому класі. Таким чином, абстрактний метод визначає інтерфейс, але не реалізацію методу. Звичайно, абстрактні класи і методи приносять відому користь, але покладений в їх основу принцип може бути розвинений далі. У C# передбачено поділ інтерфейсу класу і його реалізація за допомогою ключового слова interface.

З точки зору синтаксису інтерфейси подібні абстрактним класам. Але в інтерфейсі ні у одного з методів не повинно бути тіла. Це означає, що в інтерфейсі взагалі не надається ніякої реалізації. У ньому вказується тільки, що саме слід робити, але не як це робити. Як тільки інтерфейс буде визначено, він може бути реалізований в будь-якій кількості класів. Крім того, в одному класі може бути реалізована будь-яка кількість інтерфейсів.

Для реалізації інтерфейсу в класі повинні бути надані тіла методів, описаних в цьому інтерфейсі. Кожному класу надається повна свобода для визначення деталей своєї власної реалізації інтерфейсу. Отже, один і той же інтерфейс може бути реалізований в двох класах по-різному. Проте, в кожному з них повинен підтримуватися один і той же набір методів даного інтерфейсу. А в тому коді, де відомий такий інтерфейс, можуть використовуватися об'єкти будь-якого з цих двох класів, оскільки інтерфейс для всіх цих об'єктів залишається однаковим. Завдяки підтримці інтерфейсів в C# може бути в повній мірі реалізований головний принцип поліморфізму: один інтерфейс – безліч методів.

Інтерфейси оголошуються за допомогою ключового слова interface. Вони, по суті, є абстрактними методами. Крім методів, в інтерфейсах можна також вказувати властивості, індексатори і події. Інтерфейси не можуть містити члени даних. У них не можна також визначити конструктори, деструктори або операторні методи. Крім того, жоден з членів інтерфейсу не може бути оголошений як static.

1.1. Реалізація інтерфейсів

Як тільки інтерфейс буде визначено, він може бути реалізований в одному або декількох класах. Для реалізації інтерфейсу досить вказати його ім'я після імені класу, аналогічно базовому класу.

У класі допускається реалізовувати кілька інтерфейсів. В цьому випадку всі реалізовані в класі інтерфейси вказуються списком через кому. У класі можна успадковувати базовий клас і в той же час реалізувати один або більше інтерфейсів. У такому випадку ім'я базового класу має бути зазначено перед списком інтерфейсів, що розділяються комою.

2. Застосування інтерфейсних посилань

Як це не здасться дивним, але в C# допускається оголошувати змінні посилального інтерфейсного типу, змінні посилання на інтерфейс. Така змінна може посилатися на будь-який об'єкт, який реалізує її інтерфейс. При виклику методу для об'єкта за допомогою інтерфейсного посилання виконується його варіант, реалізований в класі даного об'єкта. Цей процес аналогічний застосуванню посилання на базовий клас для доступу до об'єкта похідного класу.

І ще одне зауваження: змінній посилання на інтерфейс доступні тільки методи, оголошені в її інтерфейсі. Тому інтерфейсне посилання не можна використовувати для доступу до будь-яких інших змінних і методів, які підтримуються об'єктом класу, що реалізує даний інтерфейс.

3. Інтерфейсні властивості

Аналогічно методам, властивості вказуються в інтерфейсі взагалі без тіла.

Очевидно, що у визначенні інтерфейсних властивостей, доступних тільки для читання або тільки для запису, повинен бути присутнім єдиний аксессор: get або set відповідно.

4. Інтерфейсні індексатори

В інтерфейсі можна також вказувати індексатори.

Як і раніше, в оголошенні інтерфейсних індексаторів, доступних тільки для читання або тільки для запису, повинен бути присутнім єдиний аксессор: get або set відповідно.

5. Спадкування інтерфейсів

Один інтерфейс може успадковувати інший. Синтаксис спадкування інтерфейсів такий же, як і у класів. Коли в класі реалізується один інтерфейс, що успадковує інший, в ньому повинні бути реалізовані всі члени, визначені в ланцюжку спадкування інтерфейсів.

6. Приховування імен при спадкуванні інтерфейсів

Коли один інтерфейс успадковує інший, то в похідному інтерфейсі може бути оголошений член, що приховує член з аналогічним ім'ям в базовому інтерфейсі. Таке приховування імен відбувається в тому випадку, якщо член в похідному інтерфейсі оголошується таким же чином, як і в базовому інтерфейсі. Але якщо не вказати в оголошенні члена похідного інтерфейсу ключове слово new, то компілятор видасть відповідне попередження.

7. Явні реалізації

При реалізації члена інтерфейсу є можливість вказати його ім'я повністю разом з ім'ям самого інтерфейсу. В цьому випадку виходить явна реалізація члена інтерфейсу, або просто явна реалізація.

Для явної реалізації інтерфейсного методу можуть бути дві причини. По-перше, коли інтерфейсний метод реалізується із зазначенням його повного імені, то такий метод виявляється доступним не за допомогою об'єктів класу, що реалізує даний інтерфейс, а по інтерфейсному посиланню. І по-друге, в одному класі можуть бути реалізовані два інтерфейси з методами, оголошеними з однаковими іменами та сигнатурами.

8. Вибір між інтерфейсом і абстрактним класом

Одна з найбільших труднощів програмування на C# складається в правильному виборі між інтерфейсом і абстрактним класом в тих випадках, коли потрібно описати функціональні можливості, але не реалізацію. У подібних випадках рекомендується дотримуватися наступних загальних правил: якщо якесь поняття можна описати з точки зору функціонального призначення, не уточнюючи конкретні деталі реалізації, то слід використовувати інтерфейс. А якщо потрібні деякі деталі реалізації, то дане поняття має бути подане абстрактним класом.

9. Стандартні інтерфейси для середовища .NET Framework

Для середовища .NET Framework визначено чимало стандартних інтерфейсів, якими можна користуватися в програмах на C#. Стандартні інтерфейси є також важливою частиною класів колекцій, що надають різні засоби, в тому числі стеки і черги, для зберігання цілих груп об'єктів.

10. Структури

Класи відносяться до посилальних типів даних. Але іноді прямий доступ до об'єктів як до значень простих типів виявляється корисним. Адже кожен доступ до об'єктів по посиланню пов'язаний з додатковими затратами на витрату обчислювальних ресурсів і оперативної пам'яті. Для вирішення подібних труднощів в C# передбачена структура, яка подібна до класу, але відноситься до типу значення, а не до посилального типу даних.

Структури оголошуються за допомогою ключового слова struct і з точки зору синтаксису подібні класам.

10.1. Про призначення структур

У зв'язку з викладеним вище виникає резонне питання: навіщо в C# включена структура, якщо вона володіє більш скромними можливостями, ніж клас. Відповідь на це питання полягає в підвищенні ефективності та продуктивності програм. Структури відносяться до типів значень, і тому ними можна оперувати безпосередньо, а не за посиланням.

11. Перерахування

Перерахування представляє собою множину іменованих цілочисельних констант. Тип даних перерахування оголошується за допомогою ключового слова enum.

Для кожної наступної символьно позначеної константи в перерахуванні задається ціле значення, яке на одиницю більше, ніж у попередньої константи. За замовчуванням значення першої символьно позначеної константи в перерахуванні дорівнює нулю.

11.1. Ініціалізація перерахування

Значення однієї або декількох символьно позначених констант в перерахуванні можна задати за допомогою ініціалізатора. Для цього достатньо вказати після символьно позначеної окремої константи знак рівності і ціле значення. Кожній наступній константі присвоюється значення, яке на одиницю більше значення попередньої ініціалізованої константи.

11.2. Вказівка базового типу перерахування

За замовчуванням в якості базового для перерахувань вибирається тип int, проте перерахування може бути створено будь-якого цілочисельного типу, за винятком char. Для того щоб вказати інший тип, окрім int, досить помістити цей тип після імені перерахування, відокремивши його двокрапкою.

11.3. Застосування перерахувань

На перший погляд перерахування можуть здатися цікавим, але не дуже необхідним елементом C#, але насправді це не так. Перерахування дуже корисні, коли в програмі потрібно одна або кілька спеціальних символьно позначених констант.

ЛЕКЦІЯ №10. СИСТЕМА ОБРОБКИ ВИНЯТКОВИХ СИТУАЦІЙ

1. Клас System.Exception
2. Основи обробки виняткових ситуацій
  2.1. Застосування пари ключових слів try і catch
  2.2. Простий приклад обробки виняткової ситуації
  2.3. Другий приклад обробки виняткової ситуації
3. Наслідки неперехвата винятків
4. Обробка виняткових ситуацій
5. Застосування декількох операторів catch
6. Перехоплення всіх винятків
7. Вкладення блоків try
8. Генерування винятків вручну
  8.1. Повторне генерування винятків
9. Використання блоку finally

Виняткова ситуація, або просто виняток, відбувається під час виконання. Використовуючи підсистему обробки виняткових ситуацій в C#, можна обробляти структурованим і контрольованим чином помилки, що виникають при виконанні програми. Головна перевага обробки виняткових ситуацій полягає в тому, що вона дозволяє автоматизувати отримання більшої частини коду, який раніше доводилося вводити в будь-яку велику програму вручну для обробки помилок. Так, якщо програма написана на мові програмування без обробки виняткових ситуацій, то при невдалому виконанні методів доводиться повертати коди помилок, які необхідно перевіряти вручну при кожному виклику методу. Це не тільки трудомісткий, але і помилко небезпечний процес.

1. Клас System.Exception

У C# винятки представлені у вигляді класів. Всі класи винятків повинні бути похідними від вбудованого в C# класу Exception, що є частиною простору імен System. Отже, всі винятки є підкласами класу Exception.

До числа найважливіших підкласів Exception відноситься клас SystemException. Саме від цього класу є похідними всі винятки, які генеруються виконуючою системою C# (тобто системою CLR). Клас SystemException нічого не додає до класу Exception, а просто визначає вершину ієрархії стандартних винятків.

У середовищі .NET Framework визначено кілька вбудованих винятків, що є похідними від класу SystemException. Наприклад, при спробі виконати поділ на нуль генерується виняток DivideByZeroException. Як буде показано далі, в C# можна створювати власні класи винятків, похідних від класу Exception.

2. Основи обробки виняткових ситуацій

Обробка виняткових ситуацій в C# організовується за допомогою чотирьох ключових слів: try, catch, throw і finally. Вони утворюють взаємопов'язану підсистему, в якій застосування одного з ключових слів має на увазі застосування іншого. Далі призначення і застосування кожного зі згаданих вище ключових слів буде розглянуто у всіх подробицях. Але спочатку необхідно дати загальне представлення про роль кожного з них у обробці виняткових ситуацій. Тому нижче коротко описаний принцип їх дії.

Оператори програми, які потрібно контролювати на появу винятків, полягають в блок try. Якщо всередині блоку try виникає виняткова ситуація, генерується виняток. Цей виняток може бути перехоплено і оброблено яким-небудь раціональним способом в коді програми за допомогою оператора, що позначається ключовим словом catch. Винятки, що виникають на рівні системи, генеруються виконуючою системою автоматично. А для генерування винятків вручну служить ключове слово throw. Будь-який код, який повинен бути неодмінно виконано після виходу з блоку try, поміщається в блок finally.

2.1. Застосування пари ключових слів try і catch

Основу обробки виняткових ситуацій в C# складає пара ключових слів try і catch. Ці ключові слова діють спільно і не можуть бути використані окремо.

Коли виняток генерується оператором try, він перехоплюється його парним оператором catch, який потім обробляє цей виняток. Залежно від типу винятку виконується і відповідний оператор catch. Так, якщо типи генерованого винятку і того, що вказується в операторі catch, збігаються, то виконується саме цей оператор, а всі інші пропускаються.

Слід, однак, мати на увазі, що якщо виняток не генерується, то блок оператора try завершується як зазвичай, і всі його оператори catch пропускаються. Виконання програми поновлюється з першого оператора, наступного після завершального оператора catch. Таким чином, оператор catch виконується лише в тому випадку, якщо генерується виняток.

2.2. Простий приклад обробки виняткової ситуації

Розглянемо простий приклад, який демонструє відстеження і перехоплювання винятку. Спроба індексувати масив за його межами призводить до помилки. Коли виникає подібна помилка, система CLR генерує виняток IndexOutOfRangeException, який визначено як стандартний для середовища .NET Framework.

2.3. Другий приклад обробки виняткової ситуації

Слід особливо підкреслити, що весь код, що виконується в блоці try, контролюється на предмет виняткових ситуацій, в тому числі і тих, які можуть виникнути в результаті виклику методу з самого блоку try. Виняток, що генерується методом в блоці try, може бути перехоплено в тому ж блоці, якщо, звичайно, цього не буде зроблено в самому методі.

В якості ще одного прикладу розглянемо наступну програму, де блок try поміщається в методі Main(). З цього блоку викликається метод GenException(), в якому і генерується виняток IndexOutOfRangeException. Цей виняток не перехоплюється методом GenException(). Але оскільки метод GenException() викликається з блоку try в методі Main(), то виняток перехоплюється в блоці catch, який пов'язаний безпосередньо з цим блоком try.

3. Наслідки неперехвата винятків

Перехоплення одного зі стандартних винятків, як в наведених вище прикладах, дає ще одну перевагу: він виключає аварійне завершення програми. Як тільки виняток буде згенеровано, він повинен бути перехоплений якимось фрагментом коду в певному місці програми. Взагалі кажучи, якщо виняток не перехоплюється в програмі, то він буде перехоплений виконуючою системою. Але справа в тому, що виконуюча система видасть повідомлення про помилку і перерве виконання програми.

4. Обробка виняткових ситуацій

Одною з головних переваг обробки виняткових ситуацій полягає в тому, що вона дозволяє вчасно відреагувати на помилку в програмі і потім продовжити її виконання. Як приклад розглянемо ще одну програму, в якій елементи одного масиву діляться на елементи іншого. Якщо при цьому відбувається поділ на нуль, то генерується виняток DivideByZeroException. Обробка подібної виняткової ситуації полягає в тому, що програма повідомляє про помилку і потім продовжує своє виконання. Таким чином, спроба поділу на нуль не призведе до аварійного завершення програми через помилку при її виконанні. Замість цього помилка обробляється витончено, не перериваючи виконання програми.

5. Застосування декількох операторів catch

З одним оператором try можна зв'язати кілька операторів catch. І на практиці це робиться досить часто. Але всі оператори catch повинні перехоплювати винятки різного типу.

Взагалі кажучи, оператори catch виконуються по порядку їх слідування в програмі. Але при цьому виконується тільки один блок catch, в якому тип винятку збігається з типом винятку, що генерується. А всі інші блоки catch пропускаються.

6. Перехоплення всіх винятків

Час від часу виникає потреба в перехопленні всіх винятків незалежно від їх типу. Для цієї мети служить оператор catch, в якому тип і змінна винятку не вказуються.

Застосовуючи універсальне перехоплення, слід мати на увазі, що його блок повинен розташовуватися останнім по порядку серед всіх блоків catch.

7. Вкладення блоків try

Один блок try може бути вкладений в інший. Виняток, що генерується у внутрішньому блоці try і не перехоплений у відповідному блоці catch, передається в зовнішній блок try.

Вкладені блоки try нерідко застосовуються для обробки різних категорій помилок різними способами. Зокрема, одні помилки вважаються невиправними і не підлягають виправленню, а інші помилки незначні і можуть бути оброблені негайно. Як правило, зовнішній блок try служить для виявлення обробки найсерйозніших помилок, а у внутрішніх блоках try обробляються менш серйозні помилки. Крім того, зовнішній блок try може стати універсальним для тих помилок, які не підлягають обробці у внутрішньому блоці.

8. Генерування винятків вручну

У наведених вище прикладах перехоплювалися винятки, що генерувалися виконуючою системою автоматично. Але винятки можуть бути згенеровані і вручну за допомогою оператора throw.

8.1. Повторне генерування винятків

Виняток, перехоплений в одному блоці catch, може бути повторно згенеровано в іншому блоці, щоб бути перехопленим в зовнішньому блоці catch. Найбільш імовірною причиною для повторного генерування винятку є надання доступу до винятку кільком обробникам. Припустимо, що один обробник оперує якимось одним аспектом винятку, а інший обробник – іншим його аспектом. Для повторного генерування винятку досить вказати оператор throw без супутнього виразу. Коли виняток генерується повторно, то він не перехоплюється знову тим же самим блоком catch, а передається в зовнішній блок catch.

9. Використання блоку finally

Іноді потрібно визначити кодовий блок, який буде виконуватися після виходу з блоку try/catch. Зокрема, виняткова ситуація може виникнути в зв'язку з помилкою, яка призводить до передчасного повернення з поточного методу. Але в цьому методі міг бути відкритий файл, який потрібно закрити, або ж встановлено мережеве з'єднання, яке потребує розривання. Подібні ситуації трапляються в програмуванні, і тому для їх вирішення в C# передбачений зручний спосіб: скористатися блоком finally.

Блок finally буде виконуватися щоразу, коли відбувається вихід з блоку try/catch, незалежно від причин, які до цього призвели. Це означає, що якщо блок try завершується нормально або через виняток, то останнім виконується код, який визначається в блоці finally. Блок finally виконується і в тому випадку, якщо будь-який код в блоці try або в пов'язаних з ним блоках catch призводить до повернення з методу.

З точки зору синтаксису блок finally розміщується після блоку try, і формально блоки catch для цього не потрібні. Блок finally можна ввести безпосередньо після блоку try, опустивши блоки catch. У цьому випадку блок finally почне виконуватися відразу ж після виходу з блоку try, але винятки розглядатись не будуть.

ЛЕКЦІЯ №11. СТАНДАРТНІ ТА КОРИСТУВАЦЬКІ ВИНЯТКИ

1. Детальний розгляд класу Exception
2. Найбільш часто використовувані винятки
3. Отримання похідних класів винятків
4. Перехоплення винятків похідних класів
5. Застосування ключових слів checked і unchecked

Обробка виняткових ситуацій важлива ще й тому, що в C# визначені стандартні винятки для типових програмних помилок, наприклад ділення на нуль або вихід індексу за межі масиву. Для реагування на подібні помилки в програмі має бути організовано відстеження та обробка відповідних виняткових ситуацій. Адже в кінцевому рахунку для успішного програмування на C# необхідно навчитися вміло користуватися підсистемою обробки виняткових ситуацій.

1. Детальний розгляд класу Exception

В операторі catch допускається вказувати тип і змінну винятку. Змінна отримує посилання на об'єкт винятку. У всіх винятках підтримуються члени, визначені в класі Exception, оскільки

всі винятки є похідними від цього класу. У цьому розділі буде розглянуто ряд найбільш корисних членів і конструкторів класу Exception і наведені конкретні приклади використання змінної винятку.

У класі Exception визначається ряд властивостей. До числа найцікавіших відносяться три властивості: Message, StackTrace і TargetSite. Всі ці властивості доступні тільки для читання. Властивість Message містить символьну строчку, що описує характер помилки; властивість StackTrace – строчку з викликами стека, що призвели до виняткової ситуації, а властивість TargetSite отримує об'єкт, що позначає метод, що згенерував виняток.

Крім того, в класі Exception визначається ряд методів. Найчастіше доводиться користуватися методом ToString(), що повертає символьну строчку з описом винятку. Цей метод автоматично викликається, наприклад, при відображенні винятку з допомогою методу WriteLine().

У класі Exception визначаються чотири наступних конструктора.

Перший конструктор використовується за умовчанням. У другому конструкторі вказується строчка повідомлення, пов'язаний з властивістю Message, яке має відношення до генерованого винятку. У третьому конструкторі вказується так званий внутрішній виняток. Цей конструктор використовується в тому випадку, коли один виняток породжує інший, причому внутрішній виняток визначає перший виняток, який буде порожнім, якщо внутрішній виняток відсутній. Якщо внутрішній виняток присутній, то він може бути отриманий з властивості InnerException, що визначається в класі Exception. І останній конструктор обробляє виняток, що відбуваються дистанційно, і тому вимагає десеріалізації.

Слід також зауважити, що в четвертому конструкторі класу Exception типи SerializationInfo і StreamingContext відносяться до простору імен System.Runtime.Serialization.

2. Найбільш часто використовувані винятки

У просторі імен System визначено кілька стандартних, вбудованих винятків. Всі ці винятки є похідними від класу SystemException, оскільки вони генеруються системою CLR при появі помилки під час виконання. Нижче перераховані деякі найбільш часто використовувані стандартні винятки.

1. ArrayTypeMismatchException – тип значення, що зберігається, несумісний з типом масиву.
2. DivideByZeroException – спроба ділення на нуль.
3. IndexOutOfRangeException – індекс виявився за межами масиву.
4. InvalidCastException – невірно виконано динамічне приведення типів.
5. OutOfMemoryException – недостатньо вільної пам'яті для подальшого виконання програми. Цей виняток може бути, наприклад, згенеровано, якщо для створення об'єкта за допомогою оператора new бракує пам'яті.
6. OverflowException – відбулося арифметичне переповнення.
7. NullReferenceException – спроба використовувати порожнє посилання, посилання, яке не вказує ні на один з об'єктів.

Більшість винятків, наведених вище, не вимагають особливих пояснень, крім винятку NullReferenceException. Цей виняток генерується при спробі використовувати порожнє посилання на неіснуючий об'єкт, наприклад, при виклику методу за порожнім посиланням. Порожнім називається таке посилання, яке не вказує ні на один з об'єктів. Для того щоб створити таке посилання, досить, наприклад, присвоїти явно пусте значення змінній типу посилання, використовуючи ключове слово null. Порожні посилання можуть також з'являтися і іншими, менш очевидними шляхами.

3. Отримання похідних класів винятків

Незважаючи на те що вбудовані винятки охоплюють найбільш поширені програмні помилки, обробка виняткових ситуацій в C# не обмежується тільки цими помилками. Насправді

одна з сильних сторін прийнятого в C# підходу до обробки виняткових ситуацій полягає в тому, що в цій мові допускається використовувати винятки, визначені користувачем, тобто тим, хто програмує на C#. Зокрема, такі спеціальні винятки можна використовувати для обробки помилок у власному коді, а створюються вони дуже просто. Для цього досить визначити клас, похідний від класу Exception. У таких класах зовсім не обов'язково щось реалізовувати – одного тільки їх існування в системі типів вже досить, щоб використовувати їх в якості винятків.

Створені користувачем класи будуть автоматично отримувати властивості і методи, визначені в класі Exception і доступні для них. Зрозуміло, будь-який з цих членів класу Exception можна перевизначити в створюваних класах винятків.

Коли створюється власний клас винятків, то, як правило, бажано, щоб в ньому підтримувалися всі конструктори, визначені в класі Exception. У простих спеціальних класах винятків цього неважко домогтися, оскільки для цього достатньо передати відповідні аргументи відповідному конструктору класу Exception, використовуючи ключове слово base. Але формально потрібно надати тільки ті конструктори, які фактично використовуються в програмі.

Розглянемо приклад програми, в якій використовується виняток спеціального типу. Розглянемо клас RangeArray, що підтримує одномірні масиви, в яких початковий і кінцевий індекси визначаються користувачем. Так, наприклад, цілком допустимим вважається масив, індексований в межах від -5 до 27. Якщо ж індекс виходив за межі масиву, то для обробки цієї помилки в класі RangeArray була визначена спеціальна змінна. Така змінна встановлювалася і перевірялася після кожної операції звернення до масиву в коді, який використовував клас RangeArray. Безумовно, такий підхід до обробки помилок незграбний і загрожує додатковими помилками. У поліпшеному варіанті класу RangeArray обробка помилок порушення меж масиву виконується більш витонченим і надійним способом за допомогою спеціально згенерованого винятку.

Коли виникає помилка порушення меж масиву класу RangeArray, генерується об'єкт типу RangeArrayException. У класі RangeArray це може статися в трьох наступних місцях: в аксесорі get індексатора, в аксесорі set індексатора і в конструкторі класу RangeArray. Для перехоплення цих винятків мається на увазі, що об'єкти типу RangeArray повинні бути сконструйовані і доступні з блоку try, що і продемонстровано у наведеній вище програмі. Використовуючи спеціальний виняток для повідомлення про помилки, клас RangeArray тепер діє як один з вбудованих в C# типів даних, і тому він може бути повністю інтегрований в механізм обробки помилок, які виявляються в програмі.

Зверніть увагу на те, що в тілі конструкторів класу винятку RangeArrayException відсутні будь-які оператори, але замість цього вони просто передають свої аргументи класу Exception, використовуючи ключове слово base. Як пояснювалося раніше, в тих випадках, коли похідний клас винятків не доповнює функції базового класу, весь процес створення винятків можна доручити конструкторам класу Exception. Адже похідний клас винятків зовсім не обов'язково повинен чимось доповнювати функції, успадковані від класу Exception.

4. Перехоплення винятків похідних класів

При спробі перехопити типи винятків, які відносяться як до базових, так і до похідних класів, слід особливо уважно дотримуватися порядку проходження операторів catch, оскільки перехоплення винятку базового класу буде збігатися з перехопленням винятку будь-яких його похідних класів. Наприклад, клас Exception є базовим для всіх винятків, і тому разом з винятком типу Exception можуть бути перехоплені і всі інші винятки похідних від нього класів. Звичайно, для більш чіткого перехоплення всіх винятків можна скористатися згадуваною раніше формою оператора catch без вказівки конкретного типу винятку. Але питання перехоплення винятку похідних класів стає вельми актуальним і в інших ситуаціях, особливо при створенні власних винятків.

Якщо потрібно перехоплювати винятки базового і похідного класів, то першим по порядку повинен слідувати оператор catch, що перехоплює виняток похідного класу. Цього правила

необхідно дотримуватися тому, що під час перехоплення винятку базового класу будуть також перехоплені винятки всіх похідних від нього класів. Правда, це правило виконується автоматично: якщо першим розташувати в коді оператор catch, що перехоплює виняток базового класу, то під час компіляції цього коду буде видано повідомлення про помилку.

Розглянемо приклад програми, де створюються два класи винятків: ExceptA і ExceptB. Клас ExceptA є похідним від класу Exception, а клас ExceptB – похідним від класу ExceptA. Потім в програмі генеруються винятки кожного типу. Заради стислості в класах спеціальних винятків надається тільки один конструктор, що приймає символьну строчку, що описує виняток. Але при розробці програм комерційного призначення в класах спеціальних винятків звичайно потрібно надавати всі чотири конструктора, що визначаються в класі Exception.

Необхідно звернути увагу на порядок проходження операторів catch. Клас ExceptB є похідним від класу ExceptA, тому виняток типу ExceptB має перехоплюватися до винятку типу ExceptA. Аналогічно, виняток типу Exception (тобто базового класу для всіх винятків) повинно перехоплюватися останнім. Для того щоб переконатися в цьому, необхідно змінити порядок проходження операторів catch. У підсумку це призведе до помилки під час компіляції.

Корисним прикладом використання оператора catch, що перехоплює виняток базового класу, служить перехоплення всієї категорії винятків. Припустимо, що створюється ряд винятків для управління деяким пристроєм. Якщо зробити їх класи похідними від загального базового класу, то в тих застосуваннях, де необов'язково з'ясовувати конкретну причину помилки, що виникла, досить перехоплювати винятки базового класу і тим самим виключити непотрібне дублювання коду.

5. Застосування ключових слів checked і unchecked

У C# є спеціальний засіб, пов'язаний з генеруванням винятків, що виникають при переповненні в арифметичних обчисленнях. Результати деяких видів арифметичних обчислень можуть перевищувати діапазон представлення чисел для типу даних, що використовується в обчисленні. В цьому випадку відбувається так зване переповнення.

У C# допускається вказувати, чи буде в коді згенеровано виняток при переповненні, за допомогою ключових слів checked і unchecked. Так, якщо потрібно вказати, що вираз буде перевірятися на переповнення, слід використовувати ключове слово checked, а якщо потрібно проігнорувати переповнення – ключове слово unchecked. В останньому випадку результат буде скорочуватися, щоб не вийти за межі діапазону представлення чисел для цільового типу виразу.

У ключового слова checked є дві загальні форми. В одній формі перевіряється конкретний вираз, і тому вона називається операторною. А в іншій формі перевіряється блок операторів, і тому вона називається блоковою.

У ключового слова unchecked також є дві загальні форми. У першій, операторній формі переповнення ігнорується при обчисленні конкретного виразу. А в другій, блоковій формі воно ігнорується при виконанні блоку операторів.

Потреба в застосуванні ключового слова checked або unchecked може виникнути, зокрема, тому, що за замовчуванням стан переповнення, що перевіряється або не перевіряється, визначається шляхом установки відповідного параметра компілятора і настройки самого середовища виконання. Тому в деяких програмах стан переповнення краще перевіряти явно.

ЛЕКЦІЯ №12. ДЕЛЕГАТИ ТА АНОНІМНІ ФУНКЦІЇ

1. Делегати
- **1.1.** Групове перетворення делегованих методів
- **1.2.** Застосування методів екземпляра в якості делегатів
- **1.3.** Групова адресація
- **1.4.** Коваріантність і контраваріантність
- **1.5.** Клас System.Delegate

1.6. Призначення делегатів
2. Анонімні функції
3. Анонімні методи
3.1. Передача аргументів анонімному методу
3.2. Повернення значення з анонімного методу
3.3. Застосування зовнішніх змінних в анонімних методах
4. Лямбда-вирази
4.1. Лямбда-оператор
4.2. Одиночні лямбда-вирази
4.3. Блочні лямбда-вирази

У цьому розділі розглядаються нові інструменти C#: делегати і лямбда-вирази. Делегат надає можливість інкапсулювати метод. Цей засіб розширює коло прикладних задач, що вирішуються при програмуванні на C#. А лямбда-вираз являє собою новий синтаксичний засіб, що забезпечує спрощений, але в той же час ефективний спосіб визначення того, що по суті є одиницею виконуваного коду. Лямбда-вирази зазвичай служать для роботи з делегатами, оскільки делегат може посилатися на лямбда-вираз. Крім того, лямбда-вирази дуже важливі для мови LINQ. В даному розділі розглядаються також анонімні методи, коваріантність, контраваріантність і групові перетворення методів.

1. Делегати

Почнемо з визначення поняття делегата. Попросту кажучи, делегат представляє собою об'єкт, який може посилатися на метод. Отже, коли створюється делегат, то в результаті виходить об'єкт, що містить посилання на метод. Більш того, метод можна викликати за цим посиланням. Іншими словами, делегат дозволяє викликати метод, на який він посилається. Нижче буде показано, наскільки дієвим виявляється такий принцип.

Слід особливо підкреслити, що один і той же делегат може бути використаний для виклику різних методів під час виконання програми, для чого достатньо змінити метод, на який посилається делегат. Таким чином, метод, що викликається делегатом, визначається під час виконання, а не в процесі компіляції. У цьому, власне, і полягає головна перевага делегата.

Тип делегата оголошується за допомогою ключового слова delegate.

1.1. Групове перетворення делегованих методів

Ще в версії C# 2.0 було впроваджено спеціальний засіб, що істотно спрощує синтаксис присвоювання методу делегату. Це так зване групове перетворення методів, що дозволяє присвоїти ім'я методу делегату, не вдаючись до оператора new або явного виклику конструктора делегата. Синтаксис групового перетворення методів істотно спрощений у порівнянні з колишнім підходом до делегування, тому далі використовується саме він.

1.2. Застосування методів екземпляра в якості делегатів

Делегат може використовувати статичні методи, так і посилатися на методи екземпляра, хоча для цього потрібно посилання на об'єкт.

1.3. Групова адресація

Одною з найбільш примітних властивостей делегата є підтримка групової адресації. Попросту кажучи, групова адресація – це можливість створити список, або ланцюжок викликів, для методів, які викликаються автоматично при зверненні до делегату. Для цього достатньо отримати екземпляр делегата, а потім додати методи в ланцюжок за допомогою оператора + або

+ =. Для видалення методу з ланцюжка служить оператор - або - =. Якщо делегат повертає значення, то їм стає значення, що повертається останнім методом в списку викликів. Тому делегат, в якому використовується групова адресація, зазвичай має повертати тип void.

1.4. Коваріантність і контраваріантність

Делегати стають ще більш гнучкими засобами програмування завдяки двом властивостям: коваріантності і контраваріантності. Як правило, метод, який передається делегату, повинен мати такий же тип, що повертається, і сигнатуру, як і делегат. Але щодо похідних типів це правило виявляється не таким суворим завдяки коваріантності і контраваріантності. Зокрема, коваріантність дозволяє присвоїти делегату метод, що повертає тип класу, похідний від класу, який зазначений в типі, що повертається в делегаті. А контраваріантність дозволяє присвоїти делегату метод, типом параметра якого служить клас, який є базовим для класу, зазначеним в оголошенні делегата.

1.5. Клас System.Delegate

Всі делегати і класи виявляються похідними неявним чином від класу System.Delegate. Як правило, членами цього класу не користуються безпосередньо, і це не робиться явним чином в даному розділі. Але члени класу System.Delegate можуть виявитися корисними в ряді особливих випадків.

1.6. Призначення делегатів

Як правило, делегати застосовуються з двох причин. По-перше, делегати підтримують події. І по-друге, делегати дозволяють викликати методи під час виконання програми, не знаючи про них нічого певного в ході компіляції. Це дуже зручно для створення базової конструкції, що допускає підключення окремих програмних компонентів. Розглянемо як приклад графічну програму, аналогічну стандартної сервісної програми Windows Paint. За допомогою делегата можна надати користувачеві можливість підключати спеціальні кольорові фільтри або аналізатори зображень. Крім того, користувач може складати з цих фільтрів або аналізаторів цілі послідовності.

2. Анонімні функції

Метод, на який посилається делегат, нерідко використовується тільки для цієї мети. Іншими словами, єдиною підставою для існування методу є та обставина, що він може бути викликаний за допомогою делегата, але сам він не викликається взагалі. У подібних випадках можна скористатися анонімною функцією, щоб не створювати окремий метод. Анонімна функція, по суті, являє собою безіменний кодовий блок, який передається конструктору делегата. Перевага анонімної функції полягає, зокрема, в її простоті. Завдяки їй відпадає необхідність оголошувати окремий метод, єдине призначення якого полягає в тому, що він передається делегату.

Починаючи з версії 3.0, в C# передбачено два різновиди анонімних функцій: анонімні методи і лямбда-вирази.

3. Анонімні методи

Анонімний метод – один із способів створення безіменного блоку коду, пов'язаного з конкретним екземпляром делегата. Для створення анонімного методу досить вказати кодовий блок після ключового слова delegate.

3.1. Передача аргументів анонімному методу

Анонімному методу можна передати один або кілька аргументів. Для цього достатньо вказати в дужках список параметрів після ключового слова delegate, а при зверненні до екземпляра делегата – передати йому відповідні аргументи.

3.2. Повернення значення з анонімного методу

Анонімний метод може повертати значення. Для цієї мети служить оператор return, діючий в анонімному методі таким же чином, як і в іменованому методі. Як і слід було очікувати, тип значення має бути сумісним з типом, що повертається, в оголошенні делегата.

3.3. Застосування зовнішніх змінних в анонімних методах

Локальна змінна, в область дії якої входить анонімний метод, називається зовнішньою змінною. Такі змінні доступні для використання в анонімному методі. І в цьому випадку зовнішня змінна вважається захопленою. Захоплена змінна існує до тих пір, поки делегат, що її захопив, не буде зібраний в сміття. Тому якщо локальна змінна, яка зазвичай припиняє своє існування після виходу з кодового блоку, використовується в анонімному методі, то вона продовжує існувати до тих пір, поки не буде знищений делегат, який посилається на цей метод.

Незважаючи на те що застосування захоплених змінних може привести до досить несподіваних результатів, воно все ж логічно обґрунтовано. Адже коли анонімний метод захоплює змінну, вона продовжує існувати до тих пір, поки використовується захоплюючий її делегат. В іншому випадку захоплена змінна виявилася б невизначеною, коли вона могла б знадобитися делегату.

4. Лямбда-вирази

Незважаючи на всю цінність анонімних методів, їм на зміну прийшов досконаліший підхід: лямбда-вираз. Не буде перебільшенням сказати, що лямбда-вираз відноситься до одних з найбільш важливих нововведень в C#, починаючи з випуску вихідної версії 1.0 цієї мови програмування. Лямбда-вираз ґрунтується на абсолютно новому синтаксичному елементі і служить більш ефективною альтернативою анонімному методу. І хоча лямбда-вирази знаходять застосування головним чином в роботі з LINQ, вони часто використовуються і разом з делегатами та подіями. Саме про це застосування лямбда-виразів і піде мова в даному розділі.

Лямбда-вираз – це інший спосіб створення анонімної функції. Отже, лямбда-вираз може бути присвоєно делегату. А оскільки лямбда-вираз вважається більш ефективним, ніж еквівалентний йому анонімний метод, то в більшості випадків рекомендується віддавати перевагу саме йому.

4.1. Лямбда-оператор

У всіх лямбда-виразах застосовується новий лямбда-оператор =>, який розділяє лямбда-вираз на дві частини. У лівій його частині вказується вхідний параметр або кілька параметрів, а в правій частині – тіло лямбда-вираза. Оператор => іноді описується такими словами, як переходить або стає. У C# підтримуються два різновиди лямбда-виразів в залежності від тіла самого лямбда-вираза.

4.2. Одиночні лямбда-вирази

В одиночному лямбда-виразі частина, яка перебуває праворуч від оператора =>, впливає на параметр або ряд параметрів, що вказується зліва. Результатом обчислення такого виразу є результат виконання лямбда-оператора.

4.3. Блочні лямбда-вирази

Другим різновидом є блочний лямбда-вираз. Для такого лямбда-виразу характерні розширені можливості виконання різних операцій, оскільки в його тілі допускається вказувати кілька операторів. Для створення блочного лямбда-виразу досить заключити тіло виразу в фігурні дужки. Крім можливості використовувати кілька операторів, в іншому блочний лямбда-вираз, практично нічим не відрізняється від щойно розглянутого одиночного лямбда-виразу.

ЛЕКЦІЯ №13. ОБРОБКА ПОДІЙ

1. Події
  1.1. Приклад групової адресації події
  1.2. Методи екземпляра в порівнянні зі статичними методами в якості обробників подій
2. Застосування аксесорів подій
3. Різноманітні можливості подій
4. Застосування анонімних методів і лямбда-виразів разом з подіями
5. Рекомендації по обробці подій в середовищі .NET Framework
6. Застосування делегатів EventHandler <TEventArgs> і EventHandler
7. Практичний приклад обробки подій

Ще одним важливим засобом C#, що базується на делегатах, є подія. Подія, по суті, являє собою автоматичне повідомлення про те, що відбулася деяка дія. Події діють за таким принципом: об'єкт, що виявляє інтерес до події, реєструє обробник цієї події. Коли ж подія відбувається, викликаються всі зареєстровані обробники цієї події. Обробники подій зазвичай представлені делегатами.

1. Події

Події є членами класу і оголошуються за допомогою ключового слова event.

Розглянемо приклад, що містить основні елементи, необхідні для обробки подій. Він починається з оголошення типу делегата для обробника подій. Всі події активізуються за допомогою делегатів. Тому тип делегата події визначає тип, що повертається, і сигнатуру для події. Далі створюється клас події. В цьому класі оголошується подія. Ключове слово event повідомляє компілятору про те, що оголошується подія. Крім того, в класі оголошується метод, що викликається для сигналізації про запуск події. Це означає, що він викликається, коли відбувається подія. У методі викликається обробник подій за допомогою делегата. Обробник викликається лише в тому випадку, якщо подія не є порожньою. А оскільки інтерес до події повинен бути зареєстрований в інших частинах програми, щоб отримувати повідомлення про нього, то метод може бути викликаний до реєстрації будь-якого обробника події. Але щоб уникнути виклику по порожньому посиланню делегат події повинен бути перевірений, щоб переконатися в тому, що він не є порожнім.

У класі створюється обробник подій. В даному простому прикладі обробник подій просто виводить повідомлення, але інші обробники можуть виконувати більш змістовні функції. Далі в методі Main() створюється об'єкт класу події, а оголошений метод реєструється як обробник цієї події, що додається в список. Обробник додається в список за допомогою оператора + =. Події підтримують тільки оператори + = і - =. І нарешті, подія запускається, що призводить до виклику всіх обробників, зареєстрованих подією.

1.1. Приклад групової адресації події

Як і делегати, події підтримують групову адресацію. Це дає можливість декільком об'єктам реагувати на повідомлення про подію.

Розглянемо приклад, побудовано два додаткових класи, в яких визначаються обробники подій, сумісні з визначеним делегатом. Тому ці обробники можуть бути також включені в ланцюжок подій. Обробники в класах не є статичними. Це означає, що спочатку повинні бути створені об'єкти кожного з цих класів, а потім в ланцюжок подій повинні бути введені обробники, пов'язані з їх екземплярами.

1.2. Методи екземпляра в порівнянні зі статичними методами в якості обробників подій

Методи екземпляра і статичні методи можуть бути використані в якості обробників подій, але між ними є одна істотна відмінність. Коли статичний метод використовується в якості обробника, повідомлення про подію поширюється на весь клас. А коли в якості обробника використовується метод екземпляра, то події адресуються конкретним екземплярам об'єктів. Отже, кожен об'єкт певного класу, якому потрібно отримати повідомлення про подію, повинен бути зареєстрований окремо. На практиці більшість обробників подій є методи екземпляра, хоча це, звичайно, залежить від конкретного застосування.

2. Застосування аксесорів подій

У наведених вище прикладах події формувалися в формі, що допускала автоматичне керування списком викликів обробників подій, включаючи додавання та видалення обробників подій зі списку. Тому управління цим списком не потрібно було організовувати вручну. Завдяки саме цій властивості такі події використовуються найчастіше. Проте організувати управління списком викликів обробників подій можна і вручну, щоб, наприклад, реалізувати спеціальний механізм збереження подій.

Для управління списком обробників подій служить розширена форма оператора event, що дозволяє використовувати аксесори подій. Ці аксесори надають засоби для управління реалізацією подібного списку.

У цю форму входять два аксесори подій: add і remove. Аксесор add викликається, коли обробник подій додається в ланцюжок подій за допомогою оператора + =. У той же час аксесор remove викликається, коли обробник подій видаляється з ланцюжка подій за допомогою оператора - =.

Коли викликається аксесор add або remove, він приймає як параметр обробник, що додається або видаляється. Як і в інших різновидах аксесорів, цей неявний параметр називається value. Реалізувавши аксесор add або remove, можна організувати спеціальну схему зберігання обробників подій. Наприклад, обробники подій можна зберігати в масиві, стеці або черзі.

Розглянемо приклад, що демонструє аксесорну форму події. У ній для зберігання обробників подій використовується масив. Цей масив складається всього з трьох елементів, тому в ланцюжку подій можна зберігати одночасно тільки три обробника.

Спочатку визначається делегат обробників подій. Потім оголошується клас. У самому його початку визначається масив обробників подій, що складається з трьох елементів. Цей масив служить для зберігання обробників подій, що додаються в ланцюжок подій. За замовчуванням елементи масиву evnt ініціалізуються порожнім значенням (null). Далі оголошується подія. В цьому оголошенні використовується розширена аксесорна форма оператора event.

Коли в ланцюжок подій додається обробник подій, викликається аксесор add, і в першому невживаному елементі масиву запам'ятовується посилання на цей обробник, що міститься в неявно заданому параметрі value. Якщо в масиві відсутні вільні елементи, то видається повідомлення про помилку. Зрозуміло, в реальному коді при переповненні списку краще згенерувати відповідне виключення. Масив складається всього з трьох елементів, тому в ньому

можна зберегти тільки три обробника подій. Коли ж обробник подій видаляється з ланцюжка подій, то викликається аксесор remove і в масиві здійснюється пошук посилання на цей обробник, переданий в якості параметра value. Якщо посилання знайдено, то відповідному елементу масиву присвоюється порожнє значення (null), а значить, обробник видаляється з ланцюжка подій.

При запуску події викликається метод. У цьому методі відбувається циклічне звернення до елементів масиву для виклику по черзі кожного обробника подій.

3. Різноманітні можливості подій

Події можуть бути визначені і в інтерфейсах. При цьому події повинні надаватися класами, що реалізують інтерфейси. Події можуть бути також визначені як абстрактні. В цьому випадку конкретну подію повинно бути реалізовано в похідному класі. Але аксесорна форма подій не може бути абстрактною. Крім того, подія може бути визначена як герметична. І нарешті, подія може бути віртуальною, її можна перевизначити в похідному класі.

4. Застосування анонімних методів і лямбда-виразів разом з подіями

Анонімні методи і лямбда-вирази особливо зручні для роботи з подіями, оскільки обробник подій найчастіше викликається тільки в коді, що реалізує механізм обробки подій. Це означає, що створювати автономний метод, як правило, немає ніяких причин. А за допомогою лямбда-виразів або анонімних методів можна істотно спростити код обробки подій.

Як згадувалося вище, лямбда-виразам тепер віддається більша перевага в порівнянні з анонімними методами. Синтаксис для використання лямбда-виразів в якості обробника подій залишається таким же, як для його застосування разом з будь-яким іншим типом делегата.

Незважаючи на те що при створенні анонімної функції перевагу слід тепер віддавати лямбда-виразам, в якості обробника подій можна як і раніше використовувати анонімний метод. Синтаксис використання анонімного методу в якості обробника подій залишається таким же, як і для його застосування разом з будь-яким іншим типом делегата.

5. Рекомендації по обробці подій в середовищі .NET Framework

У C# дозволяється формувати які завгодно різновиди подій. Але заради сумісності програмних компонентів із середовищем .NET Framework слід дотримуватися рекомендацій, встановлених для цієї мети корпорацією Microsoft. Ці рекомендації, по суті, зводяться до наступного вимогу: у обробників подій повинні бути два параметра. Перший з них – посилання на об'єкт, що формує подія, другий – параметр типу EventArgs, що містить будь-яку додаткову інформацію про подію, яка потрібна обробнику.

Сам клас EventArgs не містить поля, які можуть бути використані для передачі додаткових даних обробнику. Навпаки, EventArgs служить в якості базового класу, від якого виходить похідний клас, що містить всі необхідні поля. Проте в класі EventArgs є одне поле Empty типу static, яке є об'єктом типу EventArgs без даних.

6. Застосування делегатів EventHandler <TEventArgs> і EventHandler

В середовищі .NET Framework надається вбудований узагальнений делегат під назвою EventHandler <TEventArgs>. В даному випадку тип TEventArgs позначає тип аргументу, переданого параметру EventArgs події.

Загалом, рекомендується користуватися саме таким способом, а не визначати власний делегат. Для обробки багатьох подій параметр типу EventArgs виявляється непотрібним. Тому з метою спростити створення коду в подібних ситуаціях в середу .NET Framework впроваджений

неузагальнений делегат типу EventHandler. Він може бути використаний для оголошення обробників подій, яким не потрібна додаткова інформація про події.

7. Практичний приклад обробки подій

Події нерідко застосовуються в таких орієнтованих на обмін повідомленнями середовищах, як Windows. У подібному середовищі програма просто чекає до тих пір, поки не буде отримано конкретне повідомлення, а потім вона робить відповідну дію. Така архітектура цілком придатна для обробки подій засобами C#, оскільки дає можливість створювати обробники подій для реагування на різні повідомлення і потім просто викликати обробник при отриманні конкретного повідомлення. Так, клацання лівою кнопкою миші може бути пов'язано з подією LButtonClick. При отриманні повідомлення про натискання лівою кнопкою миші викликається метод OnLButtonClick(), і про цю подію повідомляються всі зареєстровані обробники.

Розглянемо приклад, що дає представлення про принцип, за яким діє даний підхід, на основі демонстрації розробки обробника події, пов’язаного з натисканням клавіш. Всякий раз, коли на клавіатурі натискається кнопка, запускається подія KeyPress при виклику методу OnKeyPress().

Спочатку оголошується клас KeyEventArgs, похідний від класу EventArgs, що служить для передачі повідомлення про натискання клавіші обробнику подій. Потім оголошується узагальнений делегат EventHandler, що визначає обробник подій, пов'язаний з натисканням клавіш. Ці події інкапсулюються в класі KeyEvent, де визначається подія KeyPress.

У методі Main() спочатку створюється об'єкт kevt класу KeyEvent. Потім в ланцюжок подій kevt.KeyPress додається обробник, що надається лямбда-виразом. У цьому обробнику відображається факт кожного натискання клавіші. Далі в ланцюжок подій kevt.KeyPress додається ще один обробник, що надається лямбда-виразом. У цьому обробнику підраховується кількість натиснутих клавіш.

Далі починає виконуватися цикл, в якому метод kevt.OnKeyPress() викликається при натисканні клавіші. Про цю подію повідомляються всі зареєстровані обробники подій. Після закінчення циклу відображається кількість натиснутих клавіш.

ЛЕКЦІЯ №14. УЗАГАЛЬНЕНІ КЛАСИ

1. Визначення узагальнення
2. Простий приклад узагальнень
  2.1. Розрізнення узагальнених типів за аргументами типу
  2.2. Підвищення типової безпеки за допомогою узагальнень
3. Узагальнений клас із двома параметрами типу
4. Загальна форма узагальненого класу
5. Обмежені типи
  5.1. Застосування обмеження на базовий клас
  5.2. Застосування обмеження на інтерфейс
  5.3. Застосування обмеження new() на конструктор
  5.4. Обмеження типу посилання та типу значення
6. Встановлення зв'язку між двома параметрами типу за допомогою обмеження
7. Застосування кількох обмежень
8. Отримання значення параметра типу за замовчуванням

Цей розділ присвячений узагальненням – одному з найскладніших та найефективніших засобів C#. Цікаво, що узагальнення не увійшли до початкової версії 1.0 і з'явилися лише в версії 2.0, але тепер є невід'ємною частиною мови C#. Не буде перебільшенням сказати, що використання узагальнень докорінно змінило характер C#. Це нововведення не тільки означало появу нового елемента синтаксису цієї мови, а й відкрило нові можливості для внесення численних змін та оновлень до бібліотеки класів. І хоча після запровадження узагальнень

пройшло вже кілька років, наслідки цього важливого кроку досі позначаються на розвитку C# як мови програмування.

У цьому розділі описуються синтаксис, теорія та практика застосування узагальнень, а також демонструється, яким чином узагальнення забезпечують типову безпеку в ряді випадків, які раніше вважалися складними. Після читання цього розділу можна ознайомитися з розділом, який присвячено колекціям, тому що в ньому наведено чимало прикладів застосування узагальнень у класах узагальнених колекцій.

1. Визначення узагальнення

Термін узагальнення, по суті, означає параметризований тип. Особлива роль параметризованих типів у тому, що вони дозволяють створювати класи, структури, інтерфейси, методи і делегати, у яких дані, що оброблюються, вказуються у вигляді параметра.

2. Простий приклад узагальнень

Почнемо розгляд узагальнень із простого прикладу узагальненого класу. При оголошенні класу вказується T – це ім'я параметра типу. Це ім'я служить як мітка-заповнювач конкретного типу, який вказується при створенні об'єкта класу. Отже, ім'я T використовується у класі щоразу, коли потрібен параметр типу. Зверніть увагу на те, що ім'я T обмежується кутовими дужками (< >). Цей синтаксис можна узагальнити: щоразу, коли оголошується параметр типу, він вказується у кутових дужках. Оскільки параметр типу використовується у класі, такий клас вважається узагальненим.

В оголошенні класу можна вказувати будь-яке ім'я параметра типу, але за традицією вибирається ім'я T. До інших найбільш уживаних імен параметрів типу відносяться V і E.

У C# частіше визначаються такі поняття, як відкритий та закритий типи. Відкритим типом вважається такий параметр типу або будь-який узагальнений тип, для якого аргумент типу є параметром типу або включає його в себе. А будь-який тип, який не відноситься до відкритого, вважається закритим. Сконструйованим типом вважається такий узагальнений тип, котрому надані всі аргументи типів. Якщо всі ці аргументи відносяться до закритих типів, такий тип вважається закрито сконструйованим. А якщо один або кілька аргументів типу відносяться до відкритих типів, такий тип вважається відкрито сконструйованим.

2.1. Розрізнення узагальнених типів за аргументами типу

Що стосується узагальнених типів, то слід мати на увазі, що посилання на один конкретний варіант узагальненого типу не збігається за типом з іншим варіантом того самого узагальненого типу.

2.2. Підвищення типової безпеки за допомогою узагальнень

У зв'язку з викладеним вище виникає наступне резонне питання: якщо аналогічні функціональні можливості узагальненого класу можна отримати і без узагальнень, просто вказавши об'єкт як тип даних та виконавши належне приведення типів, то яка користь від того, що клас робиться узагальненим. Відповідь на це питання полягає в тому, що узагальнення автоматично забезпечують типову безпеку всіх операцій, які стосуються класу. У ході виконання цих операцій узагальнення виключають необхідність звертатися до приведення типів та перевіряти відповідність типів у коді вручну.

3. Узагальнений клас із двома параметрами типу

У класі узагальненого типу можна вказати два або більше параметрів типу. У цьому випадку параметри типу вказуються списком через кому. А оскільки у класі, наприклад, два параметри типу, то при створенні об'єкта цього класу необхідно вказувати два відповідних аргументи типу.

4. Загальна форма узагальненого класу

Синтаксис узагальнень, може бути зведений до загальної форми в якій застосовуються знаки менше та більше.

5. Обмежені типи

У багатьох випадках відсутність обмежень на вказівку аргументів типу вважається цілком прийнятним, але іноді виявляється корисно обмежити коло типів, які можуть бути зазначені як аргумент типу.

Припустимо, що потрібно створити метод, що оперує вмістом потоку, включаючи об'єкти типу FileStream або MemoryStream. На перший погляд, така ситуація ідеально підходить для застосування узагальнень, але при цьому потрібно якимось чином гарантувати, що як аргументи типу будуть використані тільки типи потоків, але не int або будь-який інший тип. Крім того, необхідно якось повідомити компілятору про те, що методи, які визначаються в класі потоку, будуть доступні для застосування. Так, в узагальненому коді має бути певним чином відомо, що в ньому може бути викликаний метод Read().

Для виходу з подібних ситуацій в C# передбачені обмежені типи. Вказуючи параметр типу, можна накласти певне обмеження цього параметра. Це робиться за допомогою оператора where при вказівці параметра типу.

Серед усіх обмежень найчастіше застосовуються обмеження на базовий клас та інтерфейс, хоча всі вони важливі однаково. Кожне з цих обмежень розглядається далі по порядку.

5.1. Застосування обмеження на базовий клас

Обмеження на базовий клас дозволяє вказувати базовий клас, який має наслідувати аргумент типу. Обмеження на базовий клас служить двом основним цілям. По-перше, воно дозволяє використовувати в узагальненому класі члени базового класу, на які вказує дане обмеження. Накладаючи обмеження на базовий клас, компілятор знає, що це аргументи типу матимуть члени, визначені у цьому базовому класі. І по-друге, обмеження на базовий клас гарантує використання лише тих аргументів типу, які підтримують вказаний базовий клас.

5.2. Застосування обмеження на інтерфейс

Обмеження на інтерфейс дозволяє вказувати інтерфейс, який має бути реалізований аргументом типу. Це обмеження служить тим самим основним цілям, як і обмеження на базовий клас. По-перше, воно дозволяє використовувати члени інтерфейсу в узагальненому класі. І по-друге, воно гарантує використання лише тих аргументів типу, які реалізують вказаний інтерфейс. Це означає, що для будь-якого обмеження, що накладається на інтерфейс, аргумент типу повинен позначати сам інтерфейс або тип, що реалізує цей інтерфейс.

5.3. Застосування обмеження new() на конструктор

Обмеження new() на конструктор дозволяє отримувати екземпляр об'єкта узагальненого типу. Як правило, створити екземпляр параметра узагальненого типу не вдається. Але це положення змінює обмеження new(), оскільки воно вимагає, щоб аргумент типу надав конструктор без параметрів. Їм може бути конструктор, що викликається за замовчуванням і

надається автоматично, якщо конструктор, що явно визначається, відсутній або конструктор без параметрів явно оголошений користувачем. Накладаючи обмеження new(), можна викликати конструктор без параметрів для створення об'єкта.

5.4. Обмеження типу посилання та типу значення

Два інших обмеження дозволяють вказати на те, що аргумент, що позначає тип, повинен бути або типу посилання, або типу значення. Ці обмеження виявляються корисними в тих випадках, коли для узагальненого коду важливо провести різницю між типом посилання і типом значення.

З оператором where ключове слово class вказує на те, що аргумент T повинен бути типу посилання. Отже, будь-яка спроба використовувати тип значення, наприклад, int або bool, замість T призведе до помилки під час компіляції.

З оператором where ключове слово struct свідчить про те, що аргумент T має бути типу значення. Отже, будь-яка спроба використовувати тип посилання, наприклад string, замість T призведе до помилки під час компіляції. Але якщо є додаткові обмеження, то в будь-якому випадку class або struct має бути першим по порядку обмеженням, що накладається.

6. Встановлення зв'язку між двома параметрами типу за допомогою обмеження

Існує різновид обмеження на базовий клас, що дозволяє встановити зв'язок між двома параметрами типу. Оператор where повідомляє компілятору про те, що аргумент типу, прив'язаний до параметра типу V, повинен бути таким самим, як і аргумент типу, прив'язаний до параметра типу T, або успадковуватися від нього. Таке обмеження параметра типу називається неприкритим обмеженням типу.

7. Застосування кількох обмежень

З параметром типу може бути пов'язано кілька обмежень. У цьому випадку обмеження вказуються списком через кому. У цьому списку першим має бути зазначене обмеження class або struct, якщо воно є, або обмеження на базовий клас, якщо воно накладається. Вказувати обмеження class або struct одночасно з обмеженням на базовий клас не дозволяється. Далі за списком має бути обмеження на інтерфейс, а останнім по порядку – обмеження new().

8. Отримання значення параметра типу за замовчуванням

Як згадувалося вище, при написанні узагальненого коду іноді важливо провести різницю між типами значень і типами посилань. Така потреба виникає, зокрема, у разі, якщо змінній параметра типу має бути присвоєно значення за замовчуванням. Для вирішення цієї дилеми можна скористатися формою оператора default. Ця форма оператора default придатна для всіх аргументів типу, будь то типи значень або типи посилань.

ЛЕКЦІЯ №15. УЗАГАЛЬНЕНІ ТИПИ

1. Узагальнені структури
2. Створення узагальненого методу
  2.1. Виклик узагальненого методу з явно вказаними аргументами типу
  2.2. Застосування обмежень в узагальнених методах
3. Узагальнені делегати
4. Узагальнені інтерфейси
5. Порівняння екземплярів параметра типу
6. Ієрархії узагальнених класів

6.1. Застосування узагальненого базового класу
6.2. Узагальнений похідний клас
7. Перевизначення віртуальних методів в узагальненому класі
8. Перевантаження методів із кількома параметрами типу
9. Коваріантність та контраваріантність у параметрах узагальненого типу
9.1. Застосування коваріантності в узагальненому інтерфейсі
9.2. Застосування контраваріантності в узагальненому інтерфейсі
9.3. Варіантні делегати
10. Створення екземплярів об'єктів узагальнених типів
11. Деякі обмеження, властиві узагальненням

Узагальнення як мовний засіб дуже важливі тому, що вони дозволяють створювати класи, структури, інтерфейси, методи і делегати для обробки різнотипних даних з дотриманням типової безпеки. Багато алгоритмів дуже схожі за своєю логікою, незалежно від типу даних, до яких вони застосовуються. Наприклад, механізм, що підтримує чергу, залишається однаковим незалежно від того, чи призначена черга для зберігання елементів типу int, string, object або класу, що визначається користувачем. До появи узагальнень для обробки даних різних типів доводилося створювати різні варіанти одного й того ж алгоритму. А завдяки узагальненням можна спочатку розробити єдине рішення незалежно від конкретного типу даних, а потім застосувати його до обробки даних різних типів без будь-яких додаткових зусиль.

1. Узагальнені структури

У C# дозволяється створювати узагальнені структури. Синтаксис для них такий самий, як і для узагальнених класів. Як і на узагальнені класи, на узагальнені структури можуть накладатися обмеження.

2. Створення узагальненого методу

В методах, що оголошуються в узагальнених класах, може використовуватися параметр типу з даного класу, а отже, такі методи автоматично стають узагальненими по відношенню до параметра типу. Але також є можливість оголосити узагальнений метод зі своїми власними параметрами типу і навіть створити узагальнений метод, визначений у неузагальненому класі.

2.1. Виклик узагальненого методу з явно вказаними аргументами типу

У більшості випадків неявної виводимості типів виявляється достатньо для виклику узагальненого методу, проте аргументи типу можуть бути вказані явним чином.

2.2. Застосування обмежень в узагальнених методах

На аргументи узагальненого методу можна накласти обмеження, вказавши їх після переліку параметрів.

3. Узагальнені делегати

Як і методи, делегати можуть бути узагальненими. Список параметрів типу слідує безпосередньо після імені делегата. Перевага узагальнених делегатів полягає в тому, що їх допускається визначати у типізованій узагальненій формі, яку можна узгодити з будь-яким сумісним методом.

4. Узагальнені інтерфейси

Крім узагальнених класів і методів, в C# допускаються узагальнені інтерфейси. Такі інтерфейси визначаються аналогічно узагальненим класам.

5. Порівняння екземплярів параметра типу

Для порівняння двох об'єктів параметра узагальненого типу вони мають реалізовувати інтерфейс IComparable або IComparable<T> та/або інтерфейс IEquatable<T>. В обох варіантах інтерфейсу IComparable для цієї мети визначено метод CompareTo(), а в інтерфейсі IEquatable<T> – метод Equals(). Різновиди інтерфейсу IComparable призначені для застосування у випадках, коли потрібно визначити відносний порядок проходження двох об'єктів. А інтерфейс IEquatable служить для визначення рівності двох об'єктів. Всі ці інтерфейси визначені в просторі імен System і реалізовані у вбудованих в C# типах даних, включаючи int, string і double. Але їх легко реалізувати і для власних створених класів.

6. Ієрархії узагальнених класів

Узагальнені класи можуть входити до ієрархії класів аналогічно неузагальненим класам. Отже, узагальнений клас може діяти як базовий чи похідний клас. Головна відмінність між ієрархіями узагальнених і неузагальнених класів у тому, що у першому випадку аргументи типу, необхідні узагальненому базовому класу, повинні передаватися всіма похідними класами вгору по ієрархії аналогічно передачі аргументів конструктора.

6.1. Застосування узагальненого базового класу

Розглянемо приклад ієрархії, де використовується узагальнений базовий клас. У цій ієрархії похідний клас успадковує від узагальненого базового класу. Параметр типу T вказується в оголошенні похідного класу і водночас передається базовому класу. Це означає, що будь-який тип, що передається похідному класу, передаватиметься також базовому класу. В похідному класі параметр типу T може не використовуватися, а тільки передаватися вгору по ієрархії базовому класу. Це означає, що у похідному класі слід неодмінно вказувати параметри типу, потрібні його узагальненому базовому класу, навіть якщо цей похідний клас не обов'язково має бути узагальненим. В похідний клас можна вільно додавати його власні параметри типу, якщо в цьому є потреба.

6.2. Узагальнений похідний клас

Неузагальнений клас може бути цілком законно базовим для узагальненого похідного класу. Базовий клас не є узагальненим, тому аргумент типу для нього не вказується. Це означає, що параметр T, що вказується в оголошенні узагальненого похідного класу, не потрібний для вказівки базового класу і навіть не може у ньому використовуватися. Отже похідний клас успадковує від базового класу звичайним чином, без виконання якихось особливих умов.

7. Перевизначення віртуальних методів в узагальненому класі

В узагальненому класі віртуальний метод може бути перевизначений так само, як і будь-який інший метод.

8. Перевантаження методів із кількома параметрами типу

Методи, параметри яких оголошуються за допомогою параметрів типу, можуть бути перевантажені. Але правила їх перевантаження спрощуються в порівнянні з методами без

параметрів типу. Як правило, метод, у якому параметр типу служить для вказівки типу даних параметра цього методу, може бути перевантажений за умови, що сигнатури обох його варіантів відрізняються. Це означає, що обидва варіанти методу, що перевантажується, повинні відрізнятися за типом або кількістю їх параметрів. Але типові відмінності повинні визначатися не за параметром узагальненого типу, а виходячи з аргументу типу, що підставляється замість параметра типу при конструюванні цього типу. Отже, метод з параметрами типу може бути перевантажений таким чином, що він виявиться придатним не для всіх можливих випадків, хоча виглядатиме вірно.

9. Коваріантність та контраваріантність у параметрах узагальненого типу

У версії C# 4.0 можливості коваріантності та контраваріантності були розширені до параметрів узагальненого типу, що застосовуються у узагальнених інтерфейсах та делегатах.

9.1. Застосування коваріантності в узагальненому інтерфейсі

Стосовно узагальненого інтерфейсу коваріантність служить засобом, що дозволяє методу повертати тип, похідний від класу, зазначеного в параметрі типу. Параметр коваріантного типу оголошується за допомогою ключового слова out, яке випереджає ім'я цього параметра.

9.2. Застосування контраваріантності в узагальненому інтерфейсі

Стосовно узагальненого інтерфейсу контраваріантність служить засобом, що дозволяє методу використовувати аргумент, тип якого належить до базового класу, зазначеного у відповідному параметрі типу. Параметр контраваріантного типу оголошується за допомогою ключового слова in, яке випереджає ім'я цього параметра.

9.3. Варіантні делегати

Коваріантність і контраваріантність підтримується в неузагальнених делегатах відносно типів, що повертаються методами, та типів, що вказуються при оголошенні параметрів. Починаючи з версії C# 4.0, можливості коваріантності та контраваріантності були поширені і на узагальнені делегати.

10. Створення екземплярів об'єктів узагальнених типів

Коли узагальнений клас компілюється в псевдокод MSIL, він зберігає всі свої параметри типу у їх узагальненій формі. А коли конкретний екземпляр класу буде потрібний під час виконання програми, то JIT-компілятор сконструює конкретний варіант цього класу у коді, що виконується, в якому параметри типу замінюються аргументами типу. У кожному екземплярі з тими самими аргументами типу буде використовуватися той самий варіант даного класу у виконуваному коді.

11. Деякі обмеження, властиві узагальненням

Нижче наведено ряд обмежень, які слід мати на увазі при використанні узагальнень.

1. Властивості, оператори, індексатори та події не можуть бути узагальненими. Але ці елементи можуть бути використані в узагальненому класі, причому з параметрами узагальненого типу цього класу.
2. До узагальненого методу не можна застосовувати модифікатор extern.
3. Типи покажчиків не можна використовувати у аргументах типу.

4. Якщо узагальнений клас містить поле типу static, то в об'єкті кожного типу, що конструюється, має бути своя копія цього поля.

ЛЕКЦІЯ №16. МОВА ІНТЕГРОВАНИХ ЗАПИТІВ

1. Основи LINQ
  1.1. Простий запит
  1.2. Неодноразове виконання запитів
  1.3. Зв'язок між типами даних у запиті
  1.4. Загальна форма запиту
2. Відбір значень, що запитуються за допомогою оператора where
3. Сортування результатів запиту за допомогою оператора orderby
4. Детальний розгляд оператора select
5. Застосування вкладених операторів from
6. Групування результатів за допомогою оператора group
7. Продовження запиту за допомогою оператора into

Без сумніву, LINQ відноситься до одного з найцікавіших засобів мови C#. Ці засоби були впроваджені у версії C# 3.0 і виявились чи не найголовнішим його доповненням, яке полягало не тільки у внесенні абсолютно нового елемента в синтаксис C#, додаванні кількох ключових слів та наданні великих можливостей, але й в значному розширенні рамок даної мови програмування та кола задач, які він дозволяє вирішувати. Простіше кажучи, впровадження LINQ стало поворотним моментом в історії розвитку C#.

Абревіатура LINQ означає Language-Integrated Query, мову інтегрованих запитів. Це поняття охоплює низку засобів, що дозволяють витягувати інформацію з джерела даних. Витягування даних становить важливу частину багатьох програм. Наприклад, програма може отримувати інформацію зі списку замовників, шукати інформацію в каталозі продукції або отримувати доступ до облікового документа, заведеного на працівника. Як правило, така інформація зберігається в базі даних, яка існує окремо від програми. Так, каталог продукції може зберігатися у реляційній базі даних. У минулому для взаємодії з такою базою даних доводилося формувати запити на мові структурованих запитів SQL. А для доступу до інших джерел даних, наприклад у форматі XML, був потрібний окремий підхід. Отже, до версії 3.0 підтримка подібних запитів в C# була відсутня. Але це становище змінилося після впровадження LINQ.

1. Основи LINQ

В основу LINQ покладено поняття запиту, в якому визначається інформація, що отримується із джерела даних. Як тільки запит буде сформовано, його можна виконати. Це робиться, зокрема, в циклі foreach. В результаті виконання запиту виводяться його результати. Тому використання запиту можна розділити на дві основні стадії. На першій стадії запит формується, а на другій – виконується. Для звернення до джерела даних за запитом, який сформований засобами LINQ, в цьому джерелі повинен бути реалізований інтерфейс IEnumerable.

1.1. Простий запит

Розглянемо простий приклад використання LINQ: запит для отримання додатних значень, що містяться у масиві цілих значень. Для застосування засобів LINQ у вихідний текст програми слід увімкнути простір імен System.Linq. Потім у програмі оголошується масив типу int. Всі масиви в C# неявним чином перетворюються в форму інтерфейсу IEnumerable<T>. Завдяки цьому будь-який масив в C# може служити в якості джерела даних, що витягуються за запитом LINQ. Далі оголошується запит, за яким з масиву витягуються елементи лише з додатними

значеннями. Усі запити починаються з оператора from, що визначає два елемента. Першим є змінна діапазону, що приймає елементи з джерела даних. Другим елементом є джерело даних. Тип змінної діапазону виводиться із джерела даних. Далі слідує оператор where, що позначає умову, якій повинен задовольняти елемент у джерелі даних, щоб його можна було отримати за запитом. Усі запити закінчуються оператором select або group. У цьому прикладі використовується оператор select, який точно визначає, що саме має бути отримано за запитом.

1.2. Неодноразове виконання запитів

Отже, у запиті визначаються правила, за якими витягуються дані, але цього явно недостатньо для отримання результатів, оскільки запит повинен бути виконаний, причому це може бути зроблено кілька разів. Якщо ж у проміжку між спробами, що послідовно виконуються, виконати один і той же запит для якого джерело даних змінюється, то одержувані результати можуть відрізнятися.

1.3. Зв'язок між типами даних у запиті

Запит включає в себе змінні, типи яких пов'язані один з одним. До них відносяться змінна запиту, змінна діапазону та джерело даних. Дотримуватися відповідності цих типів даних дуже важливо, але в той же час нелегко – принаймні, так здається на перший погляд, тому дане питання заслуговує на більш пильну увагу.

Тип змінної діапазону повинен відповідати типу елементів, що зберігаються у джерелі даних. Отже, тип змінної діапазону залежить від типу джерела даних. Як правило, тип змінної діапазону може бути виведений засобами C#. Але виведення типів може бути здійснено за умови, що в джерелі даних реалізована форма інтерфейсу IEnumerable<T>, де T позначає тип елементів у джерелі даних. Як згадувалося вище, форма інтерфейсу IEnumerable<T> реалізується у всіх масивах, як, втім, і в багатьох інших джерелах даних. Але якщо в джерелі даних реалізований неузагальнений варіант інтерфейсу IEnumerable, то тип змінної діапазону доведеться вказувати явно. І це робиться в операторі from.

Тип об'єкта, що повертається за запитом, представляє собою екземпляр інтерфейсу IEnumerable<T>, де T – тип отриманих елементів. Отже, тип змінної запиту має бути екземпляром інтерфейсу IEnumerable<T>, а значення T повинно визначатися типом значення, що вказується в операторі select.

Коли запит виконується в циклі foreach, тип змінної кроку циклу повинен бути таким же, як і тип змінної діапазону. У попередніх прикладах тип цієї змінної вказувався як int. Але є й інша можливість: надати компілятору самому вивести тип цієї змінної, і для цього достатньо вказати її тип як var.

1.4. Загальна форма запиту

Всі запити мають загальну форму, що базується на ряді наведених в таблиці нижче контекстно-залежних ключових слів.

Таблиця. Контекстно-залежні ключові слова запиту LINQ

ascending	by	descending	equals
from	group	in	into
join	let	on	orderby
select	where		

Серед них лише наведені нижче в таблиці ключові слова використовуються на початку операторів запитів.

Таблиця. Контекстно-залежні ключові слова запиту LINQ, що використовуються на початку операторів запитів.

from	group	join	let
orderby	select	where	

Запит повинен починатися з ключового слова from і закінчуватись ключовим словом select або group. Оператор select визначає тип значення, що перераховується за запитом, а оператор group повертає дані групами, причому кожна група може перераховуватися окремо. Як випливає з наведених вище прикладів, в операторі where вказуються критерії, яким повинен задовольняти шуканий елемент, щоб бути отриманим за запитом. А решта операторів дозволяє уточнити запит. Усі вони розглядаються далі по порядку.

2. Відбір значень, що запитуються за допомогою оператора where

Оператор where служить для відбору даних, що повертаються за запитом. У найпростішій формі для відбору даних використовується єдина умова. Однак для більш ретельного відбору даних можна задати кілька умов і, зокрема, в декількох операторах where. Як правило, в умові оператора where дозволяється використовувати будь-який допустимий в C# вираз, що дає булевий результат.

3. Сортування результатів запиту за допомогою оператора orderby

Найчастіше результати запиту потребують сортування. Припустимо, що потрібно отримати список прострочених рахунків по порядку залишку на рахунку: від найбільшого до найменшого або ж список імен замовників в алфавітному порядку. Незалежно від переслідуваної мети, результати запиту можна дуже легко відсортувати, використовуючи такий засіб LINQ, як оператор orderby.

Оператор orderby можна використовувати для сортування результатів запиту за одним або декількома критеріями.

Параметр оператора orderby позначає порядок сортування за зростаючою або спадаючою з обов'язковим додаванням ключового слова ascending або descending відповідно. За замовчуванням сортування проводиться за зростаючою, тому ключове слово ascending, як правило, не вказується.

4. Детальний розгляд оператора select

Оператор select визначає конкретний тип елементів, що одержуються за запитом. Застосування оператора select не обмежується лише простою функцією повернення змінної діапазону. Він може також повертати окрему частину значення змінної діапазону, результат виконання деякої операції або перетворення змінної діапазону і навіть новий тип об'єкта, що конструюється з окремих фрагментів інформації, що отримується зі змінної діапазону. Таке перетворення вихідних даних називається проектуванням. Так, за допомогою оператора select можна сформувати будь-який необхідний тип послідовності результатів, виходячи із значень, що одержуються з джерела даних.

5. Застосування вкладених операторів from

Запит може складатися з декількох операторів from, які виявляються в цьому випадку вкладеними. Такі оператори from знаходять застосування в тих випадках, коли за запитом потрібно отримати дані з двох різних джерел.

Вкладені оператори from застосовуються також для циклічного звернення до джерела даних, що міститься в іншому джерелі даних.

6. Групування результатів за допомогою оператора group

Одним із найефективніших засобів формування запиту є оператор group, оскільки він дозволяє групувати отримані результати за ключами. Використовуючи послідовність згрупованих результатів, можна легко отримати доступ до всіх даних, пов'язаних з ключем. Завдяки цій властивості оператора group доступ до даних, організованих у послідовності зв'язаних елементів, здійснюється просто та ефективно. Оператор group є одним із двох операторів, якими може закінчуватися запит. Другим оператором, який завершує запит, є select.

Результатом виконання оператора group є послідовність, що складається з елементів типу IGrouping<TKey, TElement>, узагальненого інтерфейсу, що оголошується у просторі імен System.Linq. У цьому інтерфейсі визначено колекцію об'єктів із загальним ключем. Типом змінної запиту, що повертає групу, є IEnumerable<IGrouping<TKey, TElement>>. В інтерфейсі IGrouping визначено також доступну тільки для читання властивість Key, що повертає ключ, пов'язаний із кожною колекцією.

7. Продовження запиту за допомогою оператора into

При використанні в запиті оператора select або group іноді потрібно сформувати тимчасовий результат, який буде продовженням запиту для отримання остаточного результату. Таке продовження здійснюється за допомогою оператора into в комбінації з оператором select або group.

ЛЕКЦІЯ №17. НЕБЕЗПЕЧНИЙ КОД ТА ПОКАЖЧИКИ

1. Небезпечний код
2. Основи застосування покажчиків
3. Застосування ключового слова unsafe
4. Застосування модифікатора fixed
5. Доступ до членів структури за допомогою покажчика
6. Арифметичні операції над покажчиками
7. Порівняння покажчиків
8. Покажчики та масиви
9. Покажчики та строки
10. Багаторівнева непряма адресація
11. Масиви покажчиків
12. Створення буферів фіксованого розміру

У цьому розділі розглядається засіб мови C#, який зазвичай захоплює програмістів зненацька. Це небезпечний код. У такому коді найчастіше використовуються покажчики. Спільно з небезпечним кодом покажчики дозволяють розробляти на C# застосування, які зазвичай пов'язуються з мовою C++, високою продуктивністю та системним кодом. Більше того, завдяки включенню небезпечного коду та покажчиків до складу C# у цій мові з'явилися можливості, які відсутні в Java.

У цьому розділі розглядаються також типи, що обнуляються, визначення часткових класів і методів, буфери фіксованого розміру.

1. Небезпечний код

В C# дозволяється писати так званий небезпечний код. У цьому, дивному на перший погляд, твердженні насправді немає нічого незвичайного. Небезпечним вважається не погано написаний код, а такий код, який не може бути виконаний під повним керуванням у

загальномовному виконуючому середовищі (CLR). Результатом програмування на C# зазвичай є керований код. Тим не менш ця мова програмування допускає написання коду, який не виконується під повним керуванням у середовищі CLR. Такий некерований код не підпорядковується тим же самим засобам керування та обмеженням, що й керований код, і називається він небезпечним тому, що не можна ніяк перевірити, чи не виконує він якусь небезпечну дію. Отже, термін небезпечний зовсім не означає, що коду притаманні якісь вади. Це просто означає, що код може виконувати дії, які не підлягають контролю в керованому середовищі.

Якщо небезпечний код може спричинити ускладнення, то навіщо взагалі створювати такий код. Справа в тому, що керований код не дозволяє використовувати покажчики. У C або C++ покажчики являють собою змінні, призначені для зберігання адрес інших об'єктів, вони в якійсь мірі схожі на посилання в C#. Головна відмінність покажчика полягає в тому, що він може вказувати на будь-яку область пам'яті, тоді як посилання завжди вказує на об'єкт свого типу. Але оскільки покажчик здатний вказувати практично на будь-яку область пам'яті, то існує велика ймовірність його неправильного використання. Крім того, використовуючи покажчики, легко припуститися програмних помилок. Саме тому покажчики не підтримуються при створенні керованого коду в C#. А оскільки покажчики все-таки корисні і необхідні для деяких видів програмування, наприклад, утиліт системного рівня, в C# дозволяється створювати і використовувати їх. Але при цьому всі операції з покажчиками повинні бути позначені як небезпечні, тому що вони виконуються поза керованим середовищем.

У мові C# покажчики оголошуються і використовуються таким же чином, як і в C/C++. Головне призначення C# – створення керованого коду. А здатність цієї мови програмування підтримувати некерований код слід використовувати для вирішення лише особливого роду завдань. Це, швидше, виняток, ніж правило для програмування на C#. Фактично, для компілювання некерованого коду слід використовувати параметр компілятора /unsafe. Покажчики становлять основу небезпечного коду.

2. Основи застосування покажчиків

Покажчик представляє собою змінну, що зберігає адресу якого-небудь іншого об'єкта, наприклад іншої змінної. Так, якщо в змінній x зберігається адреса змінної y, то кажуть, що змінна x вказує на змінну y. Коли покажчик вказує на змінну, то значення цієї змінної може бути отримано або змінено за покажчиком. Такі операції із покажчиками називають непрямою адресацією.

Оголошення покажчика

Змінні-покажчики повинні бути оголошені. Це означає, що в C# не можна оголосити покажчик на об'єкт певного класу. Співвідносний тип покажчика іноді ще називають базовим. Зверніть увагу на положення знаку * в оголошенні покажчика. Він повинен слідувати після найменування типу. А name означає конкретне ім'я покажчика-змінної.

Оператори * та & в покажчиках

У покажчиках застосовуються два оператори: * та &. Оператор & є унарним і повертає адресу пам'яті свого операнда. Для унарного оператора потрібен один операнд.

Другий оператор, * є доповненням оператора &. Цей унарний оператор знаходить значення змінної, розташованої за адресою, на яку вказує його операнд. Отже, цей оператор звертається до значення змінної, на яку вказує відповідний покажчик.

3. Застосування ключового слова unsafe

Будь-який код, у якому використовуються покажчики, повинен бути позначений як небезпечний за допомогою спеціального ключового слова unsafe. Подібним чином можна

позначити конкретні типи даних, наприклад, класи та структури, члени класу, у тому числі методи та оператори або окремі кодові блоки як небезпечні.

4. Застосування модифікатора fixed

У роботі з покажчиками нерідко використовується модифікатор fixed, який запобігає видаленню керованої змінної засобами збірки сміття. Потреба в цьому виникає, наприклад, у тому випадку, якщо покажчик звертається до поля в об'єкті певного класу. А оскільки покажчику нічого не відомо про дії системи збірки сміття, то він вказуватиме не на той об'єкт, якщо видалити потрібний об'єкт.

5. Доступ до членів структури за допомогою покажчика

Покажчик може вказувати на об'єкт типу структури за умови, що структура не містить типи даних посилань. Для доступу до члена структури за допомогою покажчика слід використовувати оператор стрілки.

6. Арифметичні операції над покажчиками

Над покажчиками можна виконувати тільки чотири арифметичні операції: ++, --, + та -.

Після кожного інкрементування покажчик буде вказувати на область пам'яті, де зберігається наступний елемент його відповідного типу, а після кожного декрементування покажчик буде вказувати на область пам'яті, де зберігається попередній елемент його відповідного типу.

Якщо складати покажчики неможна, то дозволяється віднімати один покажчик з іншого, за умови, що обидва покажчики мають один і той же самий відповідний тип. Результатом такої операції виявиться кількість елементів відповідного типу, які поділяють обидва покажчика.

Крім складання та віднімання цілого числа з покажчика, а також віднімання двох покажчиків, інші арифметичні операції над покажчиками не дозволяються. Зокрема, до покажчиків неможна додавати або віднімати з них значення типу float або double. Не допускаються також арифметичні операції над покажчиками типу void*.

7. Порівняння покажчиків

Покажчики можна порівнювати за допомогою таких операторів відношення, як ==, < і >. Але для того, щоб результат порівняння покажчиків виявився змістовним, обидва покажчика повинні бути якимось чином пов'язані один з одним. Так, якщо змінні p1 і p2 є покажчиками на дві різні і не пов'язані разом змінні, то будь-яке їх порівняння, як правило, не має жодного сенсу. Але якщо змінні p1 і p2 вказують на пов'язані разом змінні, наприклад на елементи одного масиву, то їх порівняння може мати певний сенс.

8. Покажчики та масиви

У C# покажчики та масиви пов'язані один з одним. Наприклад, при вказівці імені масиву без індексу в операторі з модифікатором fixed формується покажчик на початок масиву.

Індексування покажчиків

Коли покажчик звертається до масиву, його можна індексувати як сам масив. Такий синтаксис є більш зручною в деяких випадках альтернативою арифметичним операціям над покажчиками.

9. Покажчики та строки

Символьні строки реалізовані в C# в виді об'єктів. Проте окремі символи у строчці можуть бути доступні за покажчиком. Для цього покажчику типу char* присвоюється адреса початку символьної строки в наступному операторі з модифікатором fixed.

10. Багаторівнева непряма адресація

Один покажчик може вказувати на інший, а той у свою чергу – на цільове значення. Це так звана багаторівнева непряма адресація, або застосування покажчиків на покажчики. Таке застосування покажчиків може здатися, на перший погляд, заплутаним. Значенням звичайного покажчика є адреса змінної, що містить потрібне значення. Якщо ж застосовується покажчик на покажчик, то перший з них містить адресу другого, що вказує на змінну, що містить необхідне значення.

Багаторівнева непряма адресація може бути продовжена до будь-якої межі, але потреба більш ніж у двох рівнях адресації за покажчиками виникає дуже рідко. Насправді надмірна непряма адресація дуже важко простежується і загрожує помилками.

Змінну, яка є покажчиком на покажчик, повинно бути оголошено як такову. Для цього достатньо вказати додатковий знак * після назви типу змінної.

11. Масиви покажчиків

Покажчики можуть бути організовані в масиви, як і будь-який інший тип даних.

Оператор sizeof

Під час роботи з небезпечним кодом іноді корисно знати розмір у байтах одного з вбудованих в C# типів значень. Для отримання цієї інформації служить оператор sizeof.

Оператор stackalloc

Для розподілу пам'яті, виділеної під стек, служить оператор stackalloc. Ним можна скористатися лише при ініціалізації локальних змінних.

12. Створення буферів фіксованого розміру

Ключове слово fixed знаходить ще одне застосування при створенні одновимірних масивів фіксованого розміру. У документації на C# такі масиви називаються буферами фіксованого розміру. Такі буфери завжди є членами структури. Вони призначені для створення структури, у якій містяться елементи масиву, що утворюють буфер. Коли елемент масиву включається до складу структури, у ній, зазвичай, зберігається лише посилання на цей масив. Використовуючи буфер фіксованого розміру, у структурі можна розмістити весь масив. У результаті виходить структура, придатна у тих випадках, коли важливий її розмір, як, наприклад, в багатомовному програмуванні, при погодженні даних, створених поза програмою на C#, або ж коли потрібна некерована структура, що містить масив. Але буфери фіксованого розміру можна використовувати лише у небезпечному коді.

Навчальне видання
Конспект лекцій з освітнього компонента (навчальної дисципліни) Об’єктно-орієнтоване програмування для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 125 Кібербезпека та захист інформації (частина 2) / Укл.: К.О. Трифонова. – Одеса, 2025. – 36 с.

Укладач: Трифонова К.О., ст. викл.

Підписано до друку _____. Формат 60х84/16. Папір газетний. Друк офсетний. 0,87 ум. друк. арк. 0,94 обл. - вид. арк.
Тираж 100 пр. Зам. №

Одеський національний морський університет
65029, Одеса, вул. Мечникова, 34