Кам’янець-Подільський національний університет імені Івана Огієнка

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

Навчальна дисципліна «Об’єктно-орієнтоване програмування»

ЛАБОРАТОРНА РОБОТА #0107

Тема:

Обробка винятків

Варіант №3

**Виконала**:  
студентка 1-го курсу  
групи KNms1-B24  
Некрасова Ю.Д.

**Прийняв**:  
Слободянюк О.В.

Кам’янець-Подільський – 2025

1. **Короткі теоретичні відомості**

**Види аномалій, що можуть виникати при виконанні програм.**

Ідеальних програм не існує. Тому при розробці можуть виникнути різного роду «проблеми». Наприклад, вихід за межі масиву, некоректне введення даних користувачем. У будь-якому з цих випадків програма перестає коректно працювати.

Виділяють три поширених терміни, призначеними для опису аномалій:

1. **Програмні помилки.** Так, зазвичай, називають помилки, що допускають програмісти.

List<int> lst1 = new List<int>();

lst1 = null;

lst1.Add(5);

// System.NullReferenceException: 'Object reference not set to an instance of an object.'

//lst1 was null.

2. **Користувацькі помилки**. Наприклад, введення замість числа тексту може призвести до помилки, якщо в коді не передбачена обробка таких ситуацій.

3. **Винятки**. Винятками зазвичай називаються аномалії, що виникають під час виконання, які важко, а часом і неможливо, врахувати під час програмування додатку. Прикладами винятків можуть бути спроба підключення до бази даних, яка більше не існує, відкриття пошкодженого файлу або спроба встановлення зв'язку з машиною, яка в поточний момент знаходиться в автономному режимі. У кожному з цих випадків програміст (або кінцевий користувач) мало що може зробити з подібними "винятковими" обставинами.

*Структурована обробка винятків в .NET* – це прийом, призначений для роботи з винятковими ситуаціями, які виникають під час виконання. У термінології .NET під "винятком" маються на увазі програмні помилки, некоректне введення користувачем і помилки часу виконання.

Оператор **try** вказує блок коду, що призначений для обробки помилкових ситуацій або очистки. За блоком **try** має слідувати блок **catch**, блок **finally** або обидва блоки одразу.

Блок catch виконується тоді, коли виникає помилка в блоці try. Блок finally виконується після блока try та після блока catch (якщо він присутній), забезпечуючи очистку не залежно від того, виникла помилка чи ні.

Блок catch має доступ до об’єкта Exception, який містить інформацію про помилку.

try

{

…//під час виконання цього блоку може виникнути виняток

}

catch (ExceptionA ex)

{

…//Обробити виняток типу ExceptionA

}

catch (ExceptionB ex)

{

…// Обробити виняток типу ExceptionB

}

finally

{

…//Код виконується завжди (очистка)

}

Типи винятків (наприклад): OutOfMemoryException, IndexOutOfRangeException, FormatException, OverflowException, System.Exception (всі типи винятків). Більш специфічні типи мають розміщуватися першими.

Розглянемо наступний приклад:

class Program

{

static void Main(string[] args)

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

Console.WriteLine("Кінець програми");

Console.Read();

}

}

У даному випадку відбувається ділення числа на 0, що призведе до генерації винятку. І при запуску програми в режимі відлагодження ми побачимо у Visual Studio віконце, яке повідомляє про виключення типу System.DivideByZeroException , тобто спроба ділення на нуль.

І в цьому випадку єдине, що нам залишається, це завершити виконання програми.

Щоб уникнути такого аварійного завершення програми, слід використовувати для обробки винятків конструкцію try...catch...finally . Так, перепишемо приклад так:

class Program

{

static void Main(string[] args)

{

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch

{

Console.WriteLine("Виникло виключення!");

}

finally

{

Console.WriteLine("Блок finally");

}

Console.WriteLine("Кінець програми");

Console.Read();

}

}

Ряд виняткових ситуацій можуть бути передбачені розробником. Наприклад, нехай програма передбачає введення числа та виведення його квадрата:

static void Main(string[] args)

{

Console.WriteLine("Введіть число");

int x = Int32.Parse(Console.ReadLine());

x \*= x;

Console.WriteLine("Квадрат числа: " + x);

Console.Read();

}

Якщо користувач введе не число, а рядок, інші символи, то програма випаде в помилку. З одного боку, тут саме та ситуація, коли можна застосувати блок try..catch, щоб обробити можливу помилку. Однак набагато оптимальніше було б перевірити допустимість перетворення:

static void Main(string[] args)

{

Console.WriteLine("Введіть число");

int x;

string input = Console.ReadLine();

if (Int32.TryParse(input, out x))

{

x \*= x;

Console.WriteLine("Квадрат числа: " + x);

}

else

{

Console.WriteLine("Некоректне введення");

}

Console.Read();

}

Метод Int32.TryParse() повертає true, якщо перетворення можна здійснити, і false якщо не можна. При допустимості перетворення змінна x міститиме введене число. Так, не використовуючи, try...catch можна обробити можливу виняткову ситуацію.

З погляду продуктивності використання блоків try..catch більш накладне, ніж застосування умовних конструкцій. Тому, по можливості, замість try..catch краще використовувати умовні конструкції на перевірку виняткових ситуацій.

**Генерація винятків**

Винятки можуть генеруватися або виконуючим середовищем, або кодом користувача.

Exception ex = new Exception("Сталася помилка!");

throw ex;

Клас System.Exception має багато додаткових членів, які можуть бути корисні для уточнення природи проблеми. Наприклад, властивість Data дозволяє заповнити об’єкти винятку релевантною допоміжною інформацією у форматі колекції «Словник» елементів типу «ключ-значення» (для цього потрібно підключити простір імен System.Collections за допомогою директиви using). Наприклад:

Exception ex = new Exception("Сталася помилка!");

ex.Data.Add("TimeStamp","Час виникнення помилки: "+ DateTime.Now); //мітка часу

ex.Data.Add("Cause", "Невірне значення аргументу"); // причина

throw ex;

Блок сatch потрібно оновити для забезпечення перевірки властивості Data на предмет рівності null.

catch (Exception e)

{

//За замовчуванням поле даних є порожнім, тому перевірити його на null.

Console.WriteLine("-> Custom Data:");

if (e.Data != null)

{

foreach (DictionaryEntry de in e.Data)

Console.WriteLine("-> {0}: {1}", de.Key, de.Value);

}

}

Крім загальних винятків можна генерувати також специфічні:

throw new ArgumentNullException();

Виняток можна перехопити, обробити та згенерувати знову:

try {…}

catch (Exception ex)

{

...

throw;

}

Властивості класу System.Exception:

StackTrace – рядок, що представляє всі методи, що були викликані, починаючи від джерела винятку і завершуючи блоком catch;

Message – рядок з описом помилки;

TargetSite – властивість лише для читання, повертає об’єкт MethodBase з описом низки деталей про метод, який призвів до генерації виключення.

catch (Exception e)

{

Console.WriteLine("\n\*\*\* Error! \*\*\*");

Console.WriteLine("Member name: {0}", e.TargetSite);// ім’я члена

Console.WriteLine("Class defining member: {0}",

e.TargetSite.DeclaringType);

Console.WriteLine("Member type: {0}",

e.TargetSite.MemberType);

Console.WriteLine("Message: {0}", e.Message);

Console.WriteLine("Source: {0}", e.Source);

}

\*\*\* Error! \*\*\*

Member name: Void Accelerate(Int32)

Class defining member: SimpleException.Car

Member type: Method

Message: Zippy has overheated!

Source: SimpleException

InnerException – внутрішній виняток (якщо є), що призвів до генерації зовнішнього винятку..

**Застосування конструкцій checked і unchecked.**

При виконанні арифметичних операцій з цілочисельними типами даних може виникати переповнення, тобто ситуація при якій кількість двійкових розрядів отриманого результату перевищує розрядність змінної, в яку цей результат записується.

Переповнення виникає:

* у виразах, які використовують арифметичні оператори: ++ - - (унарний) + - \* /;
* при виконанні явного перетворення цілочисельних типів.

CLR може:

* проігнорувати переповнення і відкинути старші розряди (за замовчуванням);
* згенерувати виключення OverflowException.

Можна явно задати режим контролю переповнення за допомогою ключових слів **checked** і **unchecked**.

Ключове слово **checked** задає режим контролю переповнення з генерацією винятку.

Ключове слово **unchecked** задає ігнорування виникнення переповнення.

Ключові слова checked і unchecked можуть використовуватися як оператори в рядку перетворення.

Якщо режим контролю переповнення не вказано явно, він визначається опцією компілятора /checked. Цю опцію можна задати за допомогою властивостей проекту.

1. **Умови завдання**

Зображення, що містить текст, знімок екрана, Шрифт, число

Вміст, створений ШІ, може бути неправильним.

1. **Послідовність виконання роботи**

* ***Створення користувацького винятку KilkistException****.*

Реалізовано клас, що наслідується від Exception. Призначений для генерації помилки при перевищенні максимально допустимої кількості пасажирів у літальному апараті. Містить конструктор із повідомленням помилки.

* ***Створення користувацького винятку SmugaRozgonuException***

Аналогічно створено клас для обробки помилки при спробі створити літак з недопустимою (нульовою або від’ємною) довжиною смуги розгону.

* ***Оголошення класу Pasazhyr***

Містить властивості PIB (ПІБ пасажира) та NomerMistsya (номер місця). Конструктор ініціалізує обидві властивості.

* ***Оголошення абстрактного базового класу LitalnyiAparat***

Містить спільні характеристики для літальних апаратів: назву (Nazva), максимальну кількість пасажирів (MaksPasazhyriv) та список пасажирів (pasazhiry).

Реалізовано метод DodatyPasazhyra(Pasazhyr p) з перевіркою на перевищення ліміту.

Абстрактний метод Zlit() оголошений для реалізації в похідних класах.

* ***Оголошення класу Litak, який наслідує LitalnyiAparat***

Додано властивість DovzhynaSmugy — довжина смуги розгону.  
У конструкторі перевіряється правильність значення. Якщо значення некоректне — кидається виняток SmugaRozgonuException.

Перевизначено метод Zlit(), який виводить повідомлення про зліт літака з конкретної смуги.

* ***Оголошення класу Helikopter, який також наслідує LitalnyiAparat***

Реалізовано конструктор, що ініціалізує назву та ліміт пасажирів.

Перевизначено метод Zlit(), який моделює вертикальний зліт гелікоптера.

* ***Оголошення допоміжного класу Aeroport***

Містить лише одну властивість Nazva. Може бути розширений для майбутнього розширення функціоналу (наприклад, керування літальними апаратами).

***У методі Main() здійснено такі дії:***

1. ***Спроба створити об’єкт Litak із некоректною смугою розгону***
   * Генерується виняток SmugaRozgonuException.
   * У блоці catch створюється літак з коректною довжиною смуги.
2. ***Додавання пасажирів до літака методом DodatyPasazhyra()***
   * Два перші пасажири додаються успішно.
   * Третій пасажир викликає виняток KilkistException.
   * Повідомлення про помилку виводиться на екран.
3. ***Виклик методу Zlit() для літака***
   * Виводиться повідомлення про успішний зліт літака з конкретною довжиною смуги.
4. ***Створення об’єкта Helikopter***
   * Ініціалізується назва та ліміт пасажирів.
5. ***Додавання одного пасажира до гелікоптера та зліт***
   * Викликається метод DodatyPasazhyra() — пасажир додається.
   * Метод Zlit() виводить повідомлення про вертикальний зліт.
6. **Код програми**

using System;

using System.Collections.Generic;

class KilkistException : Exception // Виняток для перевищення кількості пасажирів

{

public KilkistException(string message) : base(message) { }

}

class SmugaRozgonuException : Exception // Виняток для некоректної довжини смуги розгону

{

public SmugaRozgonuException(string message) : base(message) { }

}

class Pasazhyr // Клас Пасажир

{

public string PIB { get; set; }

public int NomerMistsya { get; set; }

public Pasazhyr(string pib, int nomer)

{

PIB = pib;

NomerMistsya = nomer;

}

}

abstract class LitalnyiAparat // Базовий клас Літальний апарат

{

public string Nazva { get; set; }

public int MaksPasazhyriv { get; set; }

protected List<Pasazhyr> pasazhiry = new();

public void DodatyPasazhyra(Pasazhyr p)

{

if (pasazhiry.Count >= MaksPasazhyriv)

throw new KilkistException("Maximum number of passengers exceeded!");

pasazhiry.Add(p);

Console.WriteLine($"Passenger {p.PIB} added.");

}

public abstract void Zlit();

}

class Litak : LitalnyiAparat // Клас Літак

{

public double DovzhynaSmugy { get; set; }

public Litak(string nazva, int maks, double smuga)

{

if (smuga <= 0)

throw new SmugaRozgonuException("Cannot create plane - invalid runway length.");

Nazva = nazva;

MaksPasazhyriv = maks;

DovzhynaSmugy = smuga;

}

public override void Zlit()

{

Console.WriteLine($"{Nazva} is taking off from a runway of {DovzhynaSmugy} meters.");

}

}

class Helikopter : LitalnyiAparat // Клас Гелікоптер

{

public Helikopter(string nazva, int maks)

{

Nazva = nazva;

MaksPasazhyriv = maks;

}

public override void Zlit()

{

Console.WriteLine($"{Nazva} is lifting off vertically.");

}

}

class Program

{

static void Main()

{

Litak litak = null;

try // Спроба створити літак з некоректною довжиною смуги

{

litak = new Litak("Boeing 737", 2, -100);

}

catch (SmugaRozgonuException ex)

{

Console.WriteLine(ex.Message);

Console.WriteLine("Retrying with valid runway length...");

litak = new Litak("Boeing 737", 2, 1500);

}

try // Додавання пасажирів

{

litak.DodatyPasazhyra(new Pasazhyr("John Smith", 1));

litak.DodatyPasazhyra(new Pasazhyr("Alice Brown", 2));

litak.DodatyPasazhyra(new Pasazhyr("Extra Guy", 3)); // перевищення

}

catch (KilkistException ex)

{

Console.WriteLine($"Error: {ex.Message}");

}

try // Зліт літака

{

litak.Zlit();

}

catch (Exception ex)

{

Console.WriteLine($"Takeoff error: {ex.Message}");

}

var heli = new Helikopter("Airbus H160", 1); // Робота з гелікоптером

try

{

heli.DodatyPasazhyra(new Pasazhyr("Pilot Anna", 1));

heli.Zlit();

}

catch (Exception ex)

{

Console.WriteLine($"Helicopter error: {ex.Message}");

}

}

}

1. **Приклад роботи програми**

Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.