Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Інформатики та інтелектуальної власності"

**Звіт**

**Лабораторна робота 4**

з дисципліни "Кросплатформне програмування"

Виконав: студент групи КН-1224С

Мульков М. В.

Перевірив:

Івашко А.В.

Харків 2025

Зміст

[Завдання 3](#_Toc193835966)

[Реалізація коду 3](#_Toc193835967)

[Алгоритм коду 3](#_Toc193835968)

[Тестування коду 3](#_Toc193835969)

[Висновок 3](#_Toc193835970)

[Додаток А 3](#_Toc193835971)

**Робота з** **класом-контейнером у .NET Framework**

**Мета роботи**: Розробити клас-контейнер для зберігання об'єктів розробленої ієрархії класів, навчитися реалізовувати базові операції над контейнером (додавання, видалення, упорядкування) без використання стандартних методів класу Array.

# Завдання

В рамках даної роботи необхідно:

Створити клас Container, який є оболонкою над масивом для зберігання об'єктів ієрархії товарів (з попередніх лабораторних робіт).

Реалізувати метод Add() для додавання об'єкта до контейнера. Передбачити автоматичне розширення масиву при необхідності.

Реалізувати метод RemoveById() для видалення об'єкта з контейнера за вказаним індексом.

Реалізувати метод Sort() для впорядкування об'єктів у контейнері. Реалізувати алгоритм сортування самостійно.

Реалізувати метод ToString() для перетворення вмісту контейнера у рядок для виведення на консоль.

Розробити інтерактивну консольну програму, яка дозволяє тестувати функціональність класу Container: додавання, видалення, сортування та виведення вмісту контейнера.

# Реалізація коду

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення класу Container

Клас Container реалізовано як оболонка над масивом items типу object[]. Приватні поля count та size використовуються для відстеження поточної кількості елементів та розміру масиву відповідно. Конструктор за замовчуванням ініціалізує масив початковим розміром в 1 елемент.

2. Реалізація методу Add()

Метод Add відповідає за додавання нового об'єкта до контейнера. Перед додаванням перевіряється, чи є місце у внутрішньому масиві (count == size). Якщо масив заповнений, створюється новий масив подвійного розміру, елементи зі старого масиву копіюються до нового, після чого внутрішній масив items посилається на цей новий, збільшений масив. Новий об'єкт додається в кінець масиву, а лічильник count інкрементується.

3. Реалізація методу RemoveById()

Метод RemoveById видаляє об'єкт за вказаним індексом \_index. Спочатку виконується перевірка на коректність індексу. Якщо індекс виходить за межі масиву, генерується виняток IndexOutOfRangeException. Для видалення елемента створюється новий масив newArray того ж розміру. Елементи з вихідного масиву копіюються до нового масиву, пропускаючи елемент за вказаним індексом. Масив items замінюється новим масивом, а count зменшується. Метод повертає видалений об'єкт.

4. Реалізація методу Sort()

Метод Sort реалізує сортування об'єктів у контейнері за властивістю Price. Використано алгоритм бульбашкового сортування. Для порівняння цін використовується допоміжний приватний метод GetPrice(object item), який за допомогою рефлексії отримує значення властивості "Price" об'єкта, якщо така властивість існує та має тип decimal. У випадку помилки під час отримання ціни, виводиться повідомлення про помилку в консоль.

5. Реалізація методу ToString()

Метод ToString перетворює вміст контейнера в рядок. Він ітерується по масиву items та викликає метод ToString() для кожного об'єкта, додаючи результат до результуючого рядка з символом нового рядка.

6. Консольна програма тестування

Консольна програма надає користувачу інтерактивне меню для тестування класу Container. Реалізовано режими автоматичної та ручної генерації об'єктів для додавання в контейнер, перегляд вмісту контейнера, сортування та видалення елементів за індексом. Для наочності виведення інформації використовується кольорове оформлення консолі. Для виведення вмісту контейнера реалізовано табличне представлення з пагінацією для зручного перегляду великої кількості об'єктів.

## Тестування коду

Тестування коду з автоматичним створенням відображено на рис. 1.

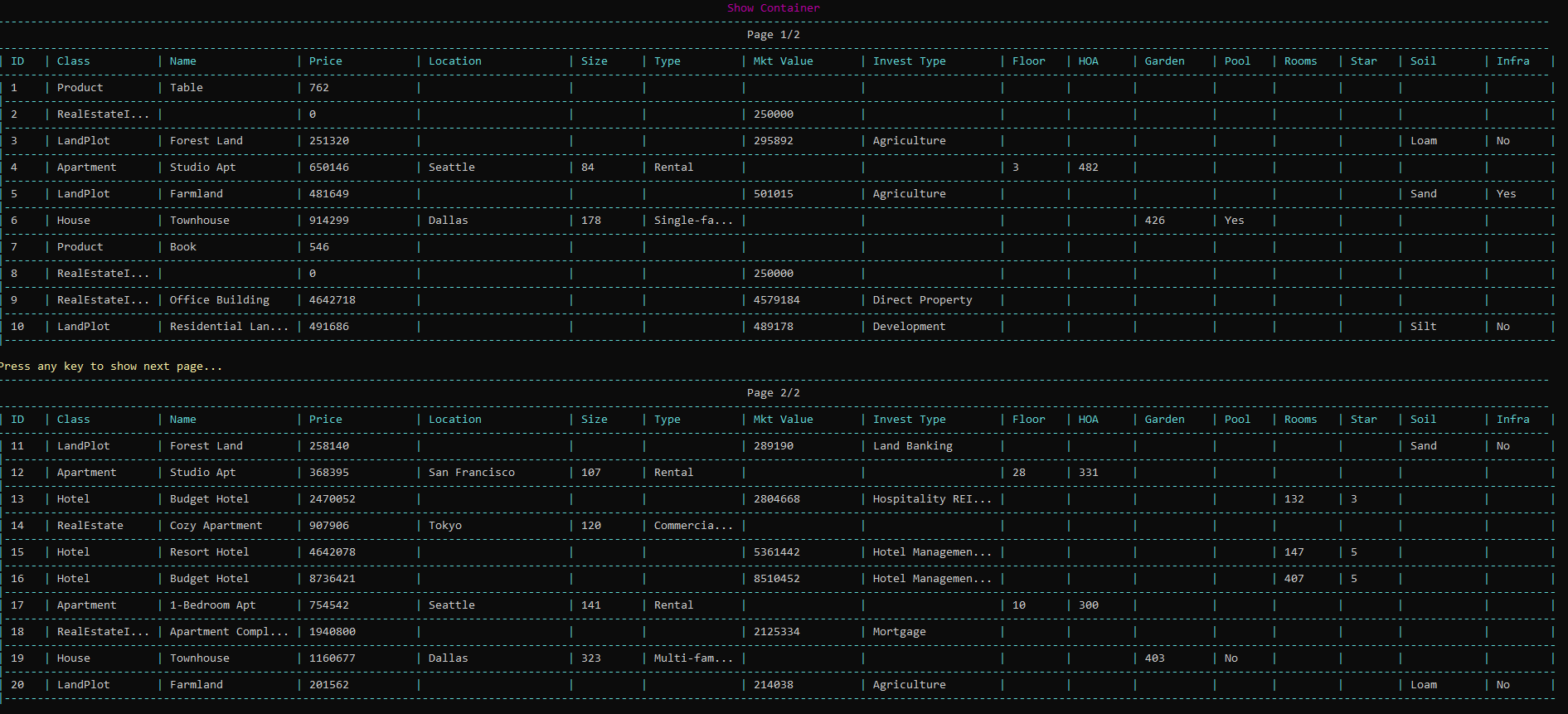


Рис. 1. Тестування коду з автоматичним створенням

Тестування коду сортування за цiною відображено на рис. 2.

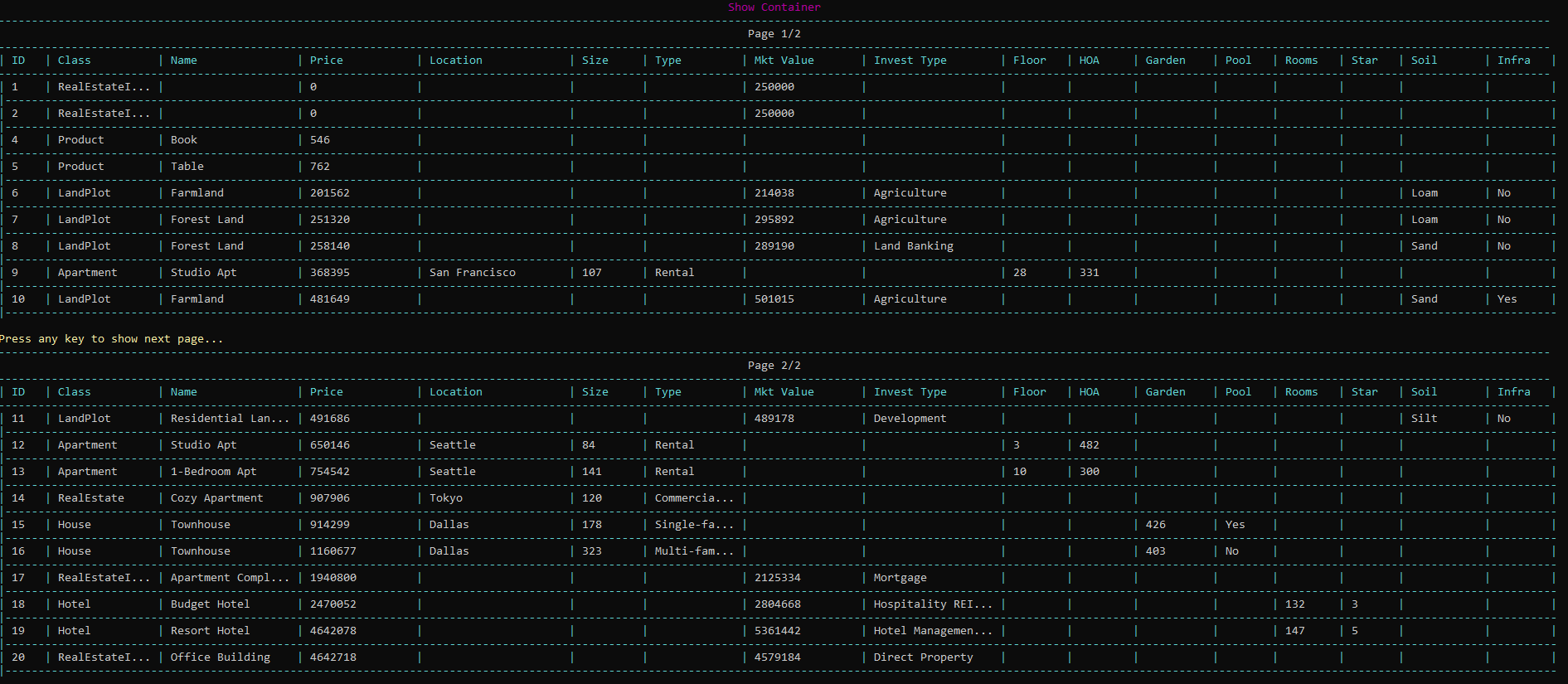


Рис. 2. Тестування коду сортування

Тестування коду видалення за iндексом відображено на рис. 3.

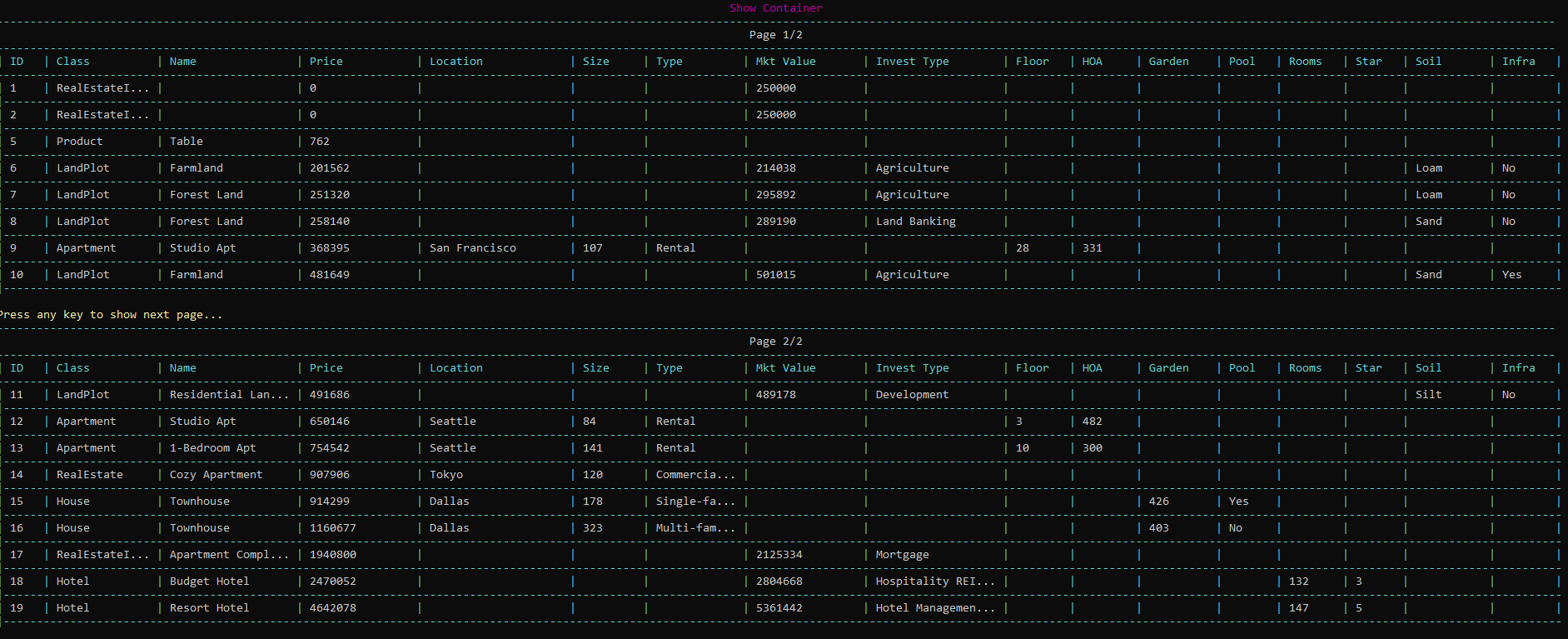


Рис. 3. Тестування коду видалення за iндексом

# Висновок

У ході виконання даної лабораторної роботи було успішно розроблено клас Container, який є фундаментальним елементом для організації зберігання та управління колекціями об'єктів у програмах. Створення цього класу, особливо з умовою ручної реалізації всіх основних операцій, стало цінним практичним досвідом, що дозволив глибше зрозуміти внутрішні механізми роботи з масивами та алгоритмами.

Реалізація методу Add продемонструвала важливість динамічного управління пам'яттю. Необхідність самостійно реалізувати розширення масиву при додаванні нових елементів дозволила на практиці усвідомити витрати на копіювання даних та вигоди від ефективного управління розміром масиву. Вибір стратегії подвоєння розміру масиву при розширенні, хоча і не єдиний можливий, є компромісом між частотою операцій розширення та витратами пам'яті.

Імплементація методу Sort з використанням алгоритму бульбашкового сортування стала практичним завданням на застосування базових алгоритмів сортування. Застосування рефлексії для отримання властивості Price та її значення продемонструвало гнучкість .NET Framework та можливість динамічно працювати з типами даних, хоча і з певними витратами на продуктивність.

Створення методу ToString для представлення вмісту контейнера у вигляді рядка підкреслило важливість надання користувачу зручного способу візуалізації даних. Реалізація табличного виведення з пагінацією в консольній програмі значно покращила зручність перегляду великих колекцій об'єктів, демонструючи важливість продуманого інтерфейсу користувача навіть у консольних застосунках.

# Додаток А

using System.Reflection;

namespace lb\_4;

class Product

{

public string Name { get; set; }

public decimal Price { get; set; }

public Product()

{

Name = string.Empty;

Price = 0;

}

public Product(string name, decimal price)

{

if (price <= 0) throw new ValueLessThanZero("Price");

Name = name;

Price = price;

}

public override string ToString()

{

return $"{Name}, Price: {Price}";

}

}

class RealEstate : Product

{

public string Location { get; set; }

public double Size { get; set; }

public string Type { get; set; }

public RealEstate()

{

Location = string.Empty;

Size = 0;

Type = string.Empty;

}

public RealEstate(string location, double size)

{

if (size <= 0) throw new ValueLessThanZero("Size");

Location = location;

Size = size;

Type = string.Empty;

}

public RealEstate(string location, double size, string type)

{

if (size <= 0) throw new ValueLessThanZero("Size");

Location = location;

Size = size;

Type = type;

}

public RealEstate(string name, decimal price, string location, double size, string type)

: base(name, price)

{

if (size <= 0) throw new ValueLessThanZero("Size");

Location = location;

Size = size;

Type = type;

}

public override string ToString()

{

return $"{Name}, Price: {Price}, Location: {Location}";

}

}

class RealEstateInvestment : Product

{

public string Location { get; set; }

public decimal MarketValue { get; set; }

public string InvestmentType { get; set; }

public RealEstateInvestment()

{

Location = string.Empty;

MarketValue = 0;

InvestmentType = string.Empty;

}

public RealEstateInvestment(string location, decimal marketValue)

{

if (marketValue <= 0) throw new ValueLessThanZero("Market value");

Location = location;

MarketValue = marketValue;

InvestmentType = string.Empty;

}

public RealEstateInvestment(string location, decimal marketValue, string investmentType)

{

if (marketValue <= 0) throw new ValueLessThanZero("Market value");

Location = location;

MarketValue = marketValue;

InvestmentType = investmentType;

}

public RealEstateInvestment(string name, decimal price, string location, decimal marketValue, string investmentType)

: base(name, price)

{

if (marketValue <= 0) throw new ValueLessThanZero("Market value");

Location = location;

MarketValue = marketValue;

InvestmentType = investmentType;

}

public override string ToString()

{

return $"{Name}, Location: {Location}, Value: {MarketValue}, Invenstment type: {InvestmentType}";

}

}

class Apartment : RealEstate

{

public int FloorNumber { get; set; }

public decimal HOAFees { get; set; }

public Apartment()

{

FloorNumber = 0;

HOAFees = 0;

}

public Apartment(int floorNumber, decimal fees)

{

if (floorNumber <= 0) throw new ValueLessThanZero("Floor number");

if (fees < 0) throw new ValueLessThanZero("Fee");

FloorNumber = floorNumber;

HOAFees = fees;

}

public Apartment(string name, decimal price, string location, double size, string type, int floorNumber, decimal fees)

: base(name, price, location, size, type)

{

if (floorNumber <= 0) throw new ValueLessThanZero("Floor number");

if (fees < 0) throw new ValueLessThanZero("Fee");

FloorNumber = floorNumber;

HOAFees = fees;

}

public override string ToString()

{

return $"{Name}, in {FloorNumber} floor, Homeowners Association fee: {HOAFees}";

}

}

class House : RealEstate

{

public double GardenSize { get; set; }

public bool Pool { get; set; }

public House()

{

GardenSize = 0;

Pool = false;

}

public House(double gardenSize, bool pool)

{

if (GardenSize < 0) throw new ValueLessThanZero("Garden size");

GardenSize = gardenSize;

Pool = pool;

}

public House(string name, decimal price, string location, double size, string type, double gardenSize, bool pool)

: base(name, price, location, size, type)

{

if (GardenSize < 0) throw new ValueLessThanZero("Garden size");

GardenSize = gardenSize;

Pool = pool;

}

public override string ToString()

{

return $"{Name}, garden size {GardenSize}, {(Pool ? "there is" : "no")} pool";

}

}

class Hotel : RealEstateInvestment

{

public int Rooms { get; set; }

public int StarRating { get; set; }

public Hotel()

{

Rooms = 0;

StarRating = 0;

}

public Hotel(int rooms, int starRating)

{

if (rooms <= 0) throw new ValueLessThanZero("Rooms");

if (starRating <= 0 || starRating > 5) throw new ValueLessThanZero("Rating", "and not higher than 5");

Rooms = rooms;

StarRating = starRating;

}

public Hotel(string name, decimal price, string location, decimal marketValue, string investmentType, int rooms, int starRating)

: base(name, price, location, marketValue, investmentType)

{

if (rooms <= 0) throw new ValueLessThanZero("Rooms");

if (starRating <= 0 || starRating > 5) throw new ValueLessThanZero("Rating", "and not higher than 5");

Rooms = rooms;

StarRating = starRating;

}

public override string ToString()

{

return $"{Name}, there are {Rooms} rooms, Hotel rating {StarRating}";

}

}

class LandPlot : RealEstateInvestment

{

public string SoilType { get; set; }

public bool InfrastructureAccess { get; set; }

public LandPlot()

{

SoilType = string.Empty;

InfrastructureAccess = true;

}

public LandPlot(string soilType, bool infrastructureAccess)

{

SoilType = soilType;

InfrastructureAccess = infrastructureAccess;

}

public LandPlot(string name, decimal price, string location, decimal marketValue, string investmentType, string soilType, bool infrastructureAccess)

: base(name, price, location, marketValue, investmentType)

{

SoilType = soilType;

InfrastructureAccess = infrastructureAccess;

}

public override string ToString()

{

return $"{Name}, Soil type {SoilType}, {(InfrastructureAccess ? "have" : "no")} access to infrastructure";

}

}

class ValueLessThanZero : Exception

{

public ValueLessThanZero(string name, string addition = "") : base(String.Format("{0} must be greater than zero {1}", name, addition)) { }

}

class Container

{

private Object[] items;

private int count;

private int size;

public Container()

{

items = new Object[1];

count = 0;

size = 1;

}

public void Add(object \_newObject)

{

if (count == size)

{

Object[] newArray = new Object[size \* 2];

for (int i = 0; i < size; i++)

{

newArray[i] = items[i];

}

items = newArray;

size \*= 2;

}

items[count] = \_newObject;

count++;

}

public object RemoveByIndex(int \_index)

{

if (\_index >= count || \_index < 0)

throw new IndexOutOfRangeException();

object deletedObject = items[\_index];

for (int i = \_index; i < count; i++)

{

items[i] = items[i + 1];

}

items[count - 1] = null;

count--;

return deletedObject;

}

public void Sort()

{

try

{

for (int i = 0; i < count - 1; i++)

{

for (int j = 0; j < count - i - 1; j++)

{

if (GetPropertyValue<decimal>(items[j], "Price") > GetPropertyValue<decimal>(items[j + 1], "Price"))

(items[j], items[j + 1]) = (items[j + 1], items[j]);

}

}

}

catch (Exception e)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine(e.Message);

Console.ResetColor();

}

}

private static T GetPropertyValue<T>(object item, string propertyName)

{

if (item == null) return default;

PropertyInfo property = item.GetType().GetProperty(propertyName);

if (property != null && property.PropertyType == typeof(T))

{

return (T)property.GetValue(item);

}

return default;

}

public string ToString()

{

string res = "";

foreach (var item in items)

{

if (item is null)

continue;

res += item.ToString() + "\n";

}

return res;

}

public Object[] GetItems()

{

return items;

}

public int GetCount()

{

return count;

}

}

class Program

{

static void Main()

{

Container container = new Container();

Random random = new Random();

while (true)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n------ Menu ------");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("1. Automatic Generation");

Console.WriteLine("2. Manual Input");

Console.WriteLine("3. Show Container");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine("#. --- ### ### ### ---");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("4. Sort Container by Price");

Console.WriteLine("5. Remove Element by Index");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine("#. --- ### ### ### ---");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("q. Exit");

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("Enter your choice: ");

Console.ResetColor();

string choice = Console.ReadLine()?.ToLower();

try

{

switch (choice)

{

case "1":

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Automatic Generation ---");

Console.ResetColor();

Console.Write("Enter number of elements to generate: ");

if (int.TryParse(Console.ReadLine(), out int count) && count > 0)

{

AutomaticGeneration(container, random, count);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nAutomatic generation of {count} elements complete.");

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid input for count (must be a positive integer). Generation cancelled.");

Console.ResetColor();

}

break;

case "2":

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Manual Input ---");

Console.ResetColor();

ManualInput(container);

break;

case "3":

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Show Container ---");

Console.ResetColor();

ShowContainer(container);

break;

case "4":

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Sorting Container by Price ---");

Console.ResetColor();

if (container.GetCount() > 0)

{

container.Sort();

Console.WriteLine("Container sorted.");

ShowContainer(container);

}

else

{

Console.WriteLine("Container is empty. Nothing to sort.");

}

break;

case "5":

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Remove Element by Index ---");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("Enter element index to remove: ");

Console.ResetColor();

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

object deletedItem = container.RemoveByIndex(index);

Console.ForegroundColor = ConsoleColor.DarkCyan;

Console.WriteLine($"Element '{deletedItem.ToString()}' was removed");

Console.ResetColor();

break;

case "q":

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Exiting...");

Console.ResetColor();

return;

default:

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid choice. Please try again.");

Console.ResetColor();

break;

}

}

catch (ValueLessThanZero ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

catch (FormatException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"\nInput Format Error: Invalid format entered. {ex.Message}");

Console.ResetColor();

}

catch (IndexOutOfRangeException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"\nAn unexpected error occurred: {ex.Message}");

Console.ResetColor();

}

finally

{

Console.ResetColor();

}

}

}

// --- Automatic Generation & Demo ---

static void AutomaticGeneration(Container container, Random random, int count)

{

Console.WriteLine("Generating elements...");

for (int i = 0; i < count; i++)

{

switch (random.Next(1, 9)) // Randomly choose a class

{

case 1: container.Add(GenerateRandomProduct(random)); break;

case 2: container.Add(GenerateRandomRealEstate(random)); break;

case 3: container.Add(GenerateRandomRealEstateInvestment(random)); break;

case 4: container.Add(GenerateRandomApartment(random)); break;

case 5: container.Add(GenerateRandomHouse(random)); break;

case 6: container.Add(GenerateRandomHotel(random)); break;

case 7: container.Add(GenerateRandomLandPlot(random)); break;

case 8: // Add some objects created with constructors having fewer params

switch (random.Next(1, 7))

{

case 1: container.Add(new RealEstate($"Loc{i}", random.Next(50, 200))); break; // Missing Type, Name, Price

case 2: container.Add(new RealEstateInvestment($"InvLoc{i}", random.Next(10000, 50000))); break; // Missing InvestmentType, Name, Price

case 3: container.Add(new Apartment(random.Next(1, 10), random.Next(50, 300))); break; // Missing Name, Price, Location, Size, Type

case 4: container.Add(new House(random.Next(100, 500), random.Next(2) == 0)); break; // Missing Name, Price, Location, Size, Type

case 5: container.Add(new Hotel(random.Next(20, 100), random.Next(1, 6))); break; // Missing Name, Price, Location, MarketValue, InvestmentType

case 6: container.Add(new LandPlot($"Soil{i}", random.Next(2) == 0)); break; // Missing Name, Price, Location, MarketValue, InvestmentType

}

break;

}

Console.Write("."); // Progress indicator

}

Console.WriteLine(); // New line after progress dots

}

// --- Manual Input ---

static void ManualInput(Container container)

{

Console.WriteLine("Choose class to create:");

Console.WriteLine("1. Product");

Console.WriteLine("2. RealEstate");

Console.WriteLine("3. RealEstateInvestment");

Console.WriteLine("4. Apartment");

Console.WriteLine("5. House");

Console.WriteLine("6. Hotel");

Console.WriteLine("7. LandPlot");

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("Enter your choice: ");

Console.ResetColor();

string classChoice = Console.ReadLine();

object newItem = null;

try

{

switch (classChoice)

{

case "1": newItem = CreateManualProduct(); break;

case "2": newItem = CreateManualRealEstate(); break;

case "3": newItem = CreateManualRealEstateInvestment(); break;

case "4": newItem = CreateManualApartment(); break;

case "5": newItem = CreateManualHouse(); break;

case "6": newItem = CreateManualHotel(); break;

case "7": newItem = CreateManualLandPlot(); break;

default:

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid class choice.");

Console.ResetColor();

return;

}

}

catch (ValueLessThanZero ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

catch (FormatException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Invalid input format: {ex.Message}");

Console.ResetColor();

}

if (newItem != null)

{

container.Add(newItem);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"{newItem.GetType().Name} added successfully.");

Console.ResetColor();

}

}

// --- Show Container (Table Display) ---

static void ShowContainer(Container container)

{

int currentCount = container.GetCount();

string title = $"Container Contents ({currentCount} items)";

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Magenta;

if (currentCount > 0)

Console.WriteLine(CenterString(title, tableWidth));

else

Console.WriteLine(title);

Console.ResetColor();

if (currentCount == 0)

{

Console.WriteLine("Container is empty.");

return;

}

int itemsPerPage = 10; // Adjust as needed

int pageCount = (int)Math.Ceiling((double)currentCount / itemsPerPage);

for (int page = 0; page < pageCount; page++)

{

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

string pageTitle = $"Page {page + 1}/{pageCount}";

Console.WriteLine(CenterString(pageTitle, tableWidth));

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

Object[] items = container.GetItems(); // Get the raw array

int startIdx = page \* itemsPerPage;

int endIdx = Math.Min((page + 1) \* itemsPerPage, currentCount);

for (int i = startIdx; i < endIdx; i++)

{

var item = items[i]; // Using the indexer here!

if (item == null) continue; // Should not happen if index is valid

WriteDataRow(i + 1, item); // Pass 1-based ID

Console.ForegroundColor = ConsoleColor.Cyan;

Console.Write("|");

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

if (page + 1 < pageCount)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"\nShowing items {startIdx + 1}-{endIdx}. Press any key for next page...");

Console.ResetColor();

Console.ReadKey();

}

}

}

// --- Helper methods for ShowContainer ---

const int idWidth = 4;

const int classWidth = 14; // Increased slightly

const int nameWidth = 18;

const int priceWidth = 15;

const int locationWidth = 20;

const int sizeWidth = 8;

const int typeWidth = 12;

const int marketValueWidth = 15;

const int investmentTypeWidth = 18;

const int floorWidth = 7;

const int hoaWidth = 7; // Increased slightly

const int gardenWidth = 9;

const int poolWidth = 6;

const int roomsWidth = 7;

const int starWidth = 6;

const int soilWidth = 10;

const int infraWidth = 7;

static int CalculateTableWidth()

{

return idWidth + classWidth + nameWidth + priceWidth + locationWidth + sizeWidth + typeWidth + marketValueWidth + investmentTypeWidth + floorWidth + hoaWidth + gardenWidth + poolWidth + roomsWidth + starWidth + soilWidth + infraWidth

+ 51; // Account for " | " spacing around each separator

}

static void WriteHeaderRow()

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.Write($"| {"ID".PadRight(idWidth)} | {"Class".PadRight(classWidth)} | {"Name".PadRight(nameWidth)} | {"Price".PadRight(priceWidth)} | {"Location".PadRight(locationWidth)} | {"Size".PadRight(sizeWidth)} | {"Type".PadRight(typeWidth)} | {"Mkt Value".PadRight(marketValueWidth)} | {"Invest Type".PadRight(investmentTypeWidth)} | {"Floor".PadRight(floorWidth)} | {"HOA Fee".PadRight(hoaWidth)} | {"GardenSz".PadRight(gardenWidth)} | {"Pool".PadRight(poolWidth)} | {"Rooms".PadRight(roomsWidth)} | {"Star".PadRight(starWidth)} | {"Soil".PadRight(soilWidth)} | {"Infra".PadRight(infraWidth)} |\n");

Console.ResetColor();

}

static void WriteDataRow(int id, object item)

{

// Helper to format nullable decimals/doubles - These remain unchanged

string FormatDecimal(decimal? d) => d?.ToString("N2") ?? "";

string FormatDouble(double? d) => d?.ToString("N1") ?? "";

string FormatBool(bool? b) => b.HasValue ? (b.Value ? "Yes" : "No") : "";

string FormatInt(int? i) => i?.ToString() ?? "";

// --- Get Values Safely ---

Type itemType = item.GetType(); // Get type once for efficiency

string name = GetPropertyValue<string>(item, "Name"); // String works fine as is

string formattedPrice = "";

if (itemType.GetProperty("Price") != null) // Check if property exists

{

decimal priceValue = GetPropertyValue<decimal>(item, "Price"); // Get non-nullable

formattedPrice = FormatDecimal(priceValue); // Pass non-nullable, implicitly converts to decimal?

}

string location = GetPropertyValue<string>(item, "Location");

string formattedSize = "";

if (itemType.GetProperty("Size") != null)

{

double sizeValue = GetPropertyValue<double>(item, "Size");

formattedSize = FormatDouble(sizeValue);

}

string type = GetPropertyValue<string>(item, "Type");

string formattedMarketValue = "";

if (itemType.GetProperty("MarketValue") != null)

{

decimal marketValueValue = GetPropertyValue<decimal>(item, "MarketValue");

formattedMarketValue = FormatDecimal(marketValueValue);

}

string investmentType = GetPropertyValue<string>(item, "InvestmentType");

string formattedFloorNumber = "";

if (itemType.GetProperty("FloorNumber") != null)

{

int floorNumberValue = GetPropertyValue<int>(item, "FloorNumber");

formattedFloorNumber = FormatInt(floorNumberValue);

}

string formattedHoaFees = "";

if (itemType.GetProperty("HOAFees") != null)

{

decimal hoaFeesValue = GetPropertyValue<decimal>(item, "HOAFees");

formattedHoaFees = FormatDecimal(hoaFeesValue);

}

string formattedGardenSize = "";

if (itemType.GetProperty("GardenSize") != null)

{

double gardenSizeValue = GetPropertyValue<double>(item, "GardenSize");

formattedGardenSize = FormatDouble(gardenSizeValue);

}

string formattedPool = "";

if (itemType.GetProperty("Pool") != null)

{

bool poolValue = GetPropertyValue<bool>(item, "Pool");

formattedPool = FormatBool(poolValue);

}

string formattedRooms = "";

if (itemType.GetProperty("Rooms") != null)

{

int roomsValue = GetPropertyValue<int>(item, "Rooms");

formattedRooms = FormatInt(roomsValue);

}

string formattedStarRating = "";

if (itemType.GetProperty("StarRating") != null)

{

int starRatingValue = GetPropertyValue<int>(item, "StarRating");

formattedStarRating = FormatInt(starRatingValue);

}

string soilType = GetPropertyValue<string>(item, "SoilType");

string formattedInfrastructureAccess = "";

if (itemType.GetProperty("InfrastructureAccess") != null)

{

bool infrastructureAccessValue = GetPropertyValue<bool>(item, "InfrastructureAccess");

formattedInfrastructureAccess = FormatBool(infrastructureAccessValue);

}

Console.Write($"| {id.ToString().PadRight(idWidth)} ");

Console.Write($"| {Truncate(itemType.Name, classWidth).PadRight(classWidth)} "); // Use cached type

Console.Write($"| {Truncate(name, nameWidth).PadRight(nameWidth)} ");

Console.Write($"| {formattedPrice.PadRight(priceWidth)} "); // Use formatted value

Console.Write($"| {Truncate(location, locationWidth).PadRight(locationWidth)} ");

Console.Write($"| {formattedSize.PadRight(sizeWidth)} "); // Use formatted value

Console.Write($"| {Truncate(type, typeWidth).PadRight(typeWidth)} ");

Console.Write($"| {formattedMarketValue.PadRight(marketValueWidth)} "); // Use formatted value

Console.Write($"| {Truncate(investmentType, investmentTypeWidth).PadRight(investmentTypeWidth)} ");

Console.Write($"| {formattedFloorNumber.PadRight(floorWidth)} "); // Use formatted value

Console.Write($"| {formattedHoaFees.PadRight(hoaWidth)} "); // Use formatted value

Console.Write($"| {formattedGardenSize.PadRight(gardenWidth)} "); // Use formatted value

Console.Write($"| {formattedPool.PadRight(poolWidth)} "); // Use formatted value

Console.Write($"| {formattedRooms.PadRight(roomsWidth)} "); // Use formatted value

Console.Write($"| {formattedStarRating.PadRight(starWidth)} "); // Use formatted value

Console.Write($"| {Truncate(soilType, soilWidth).PadRight(soilWidth)} ");

Console.Write($"| {formattedInfrastructureAccess.PadRight(infraWidth)} "); // Use formatted value

Console.WriteLine("|"); // End of row

}

static void DrawHorizontalLine(int tableWidth)

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine(new string('-', tableWidth));

Console.ResetColor();

}

static string CenterString(string s, int width)

{

if (s.Length >= width) return s;

int padding = (width - s.Length) / 2;

return new string(' ', padding) + s + new string(' ', width - s.Length - padding);

}

static string Truncate(string value, int maxLength)

{

if (string.IsNullOrEmpty(value)) return "";

return value.Length <= maxLength ? value : value.Substring(0, maxLength - 3) + "...";

}

// --- Generic Property Getter (using Reflection) ---

private static T GetPropertyValue<T>(object item, string propertyName)

{

if (item == null) return default;

PropertyInfo property = item.GetType().GetProperty(propertyName);

if (property != null && (property.PropertyType == typeof(T) || Nullable.GetUnderlyingType(property.PropertyType) == typeof(T)))

{

try

{

object value = property.GetValue(item);

if (value == null && Nullable.GetUnderlyingType(typeof(T)) != null)

{

return default(T); // Return default for nullable type (which is null)

}

if (value == null && !typeof(T).IsValueType) // Handle null for reference types

{

return default(T);

}

return (T)value;

}

catch

{

return default;

}

}

return default;

}

// --- Random Generators (Copied from original for completeness) ---

static Product GenerateRandomProduct(Random random)

{

string[] names = { "Table", "Chair", "Lamp", "Phone", "Book", "Laptop", "Mug" };

decimal price = random.Next(10, 1000) + (decimal)random.NextDouble();

return new Product(names[random.Next(names.Length)] + random.Next(100), Math.Round(price, 2));

}

static RealEstate GenerateRandomRealEstate(Random random)

{

string[] names = { "Cozy Apt", "Luxury Villa", "Small House", "Big Mansion", "Downtown Loft" };

string[] locations = { "New York", "London", "Paris", "Tokyo", "Kyiv", "Berlin", "Sydney" };

string[] types = { "Residential", "Commercial", "Industrial", "Mixed-Use" };

decimal price = random.Next(100000, 1000000) + (decimal)random.NextDouble() \* 1000;

double size = random.Next(50, 500) + random.NextDouble() \* 10;

return new RealEstate(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(size, 1), types[random.Next(types.Length)]);

}

static RealEstateInvestment GenerateRandomRealEstateInvestment(Random random)

{

string[] names = { "Office Bldg", "Shopping Mall", "Warehouse", "Apt Complex", "Data Center" };

string[] locations = { "Chicago", "Los Angeles", "Houston", "Phoenix", "Philadelphia", "Dallas" };

string[] investmentTypes = { "REIT", "Direct Prop", "Mortgage Fund", "Syndication" };

decimal price = random.Next(500000, 5000000) + (decimal)random.NextDouble() \* 10000;

decimal marketValue = price \* (decimal)(0.8 + random.NextDouble() \* 0.4); // 80% to 120% of price

return new RealEstateInvestment(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(marketValue, 2), investmentTypes[random.Next(investmentTypes.Length)]);

}

static Apartment GenerateRandomApartment(Random random)

{

string[] names = { "Studio Apt", "1BR Apt", "2BR Apt", "Penthouse", "Garden Apt" };

string[] locations = { "Miami", "San Francisco", "Seattle", "Boston", "Denver", "Austin" };

string[] types = { "Condo", "Co-op", "Rental Unit", "Loft" };

decimal price = random.Next(200000, 800000) + (decimal)random.NextDouble() \* 500;

double size = random.Next(40, 150) + random.NextDouble() \* 5;

int floorNumber = random.Next(1, 30);

decimal hoaFees = random.Next(100, 500) + (decimal)random.NextDouble() \* 50;

return new Apartment(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(size, 1), types[random.Next(types.Length)], floorNumber, Math.Round(hoaFees, 2));

}

static House GenerateRandomHouse(Random random)

{

string[] names = { "Bungalow", "Townhouse", "Ranch", "Cottage", "Colonial" };

string[] locations = { "Atlanta", "Dallas", "San Diego", "Orlando", "Las Vegas", "Nashville" };

string[] types = { "Single-family", "Multi-family", "Duplex" };

decimal price = random.Next(300000, 1200000) + (decimal)random.NextDouble() \* 1000;

double size = random.Next(100, 400) + random.NextDouble() \* 15;

double gardenSize = random.Next(0, 1000) + random.NextDouble() \* 100;

bool pool = random.Next(3) == 0; // 1 in 3 chance of pool

return new House(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(size, 1), types[random.Next(types.Length)], Math.Round(gardenSize, 1), pool);

}

static Hotel GenerateRandomHotel(Random random)

{

string[] names = { "Luxury Hotel", "Budget Inn", "Resort & Spa", "Boutique Hotel", "Airport Motel" };

string[] locations = { "Hawaii", "Bali", "Maldives", "Fiji", "Santorini", "Las Vegas Strip" };

string[] investmentTypes = { "Hospitality REIT", "Hotel Mgmt", "Timeshare", "Franchise" };

decimal price = random.Next(1000000, 10000000) + (decimal)random.NextDouble() \* 50000;

decimal marketValue = price \* (decimal)(0.9 + random.NextDouble() \* 0.3); // 90% to 120% of price

int rooms = random.Next(50, 500);

int starRating = random.Next(1, 6); // Allow 1-5 stars

return new Hotel(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(marketValue, 2), investmentTypes[random.Next(investmentTypes.Length)], rooms, starRating);

}

static LandPlot GenerateRandomLandPlot(Random random)

{

string[] names = { "Farmland", "Forest", "Comm Land", "Resid Land", "Waterfront" };

string[] locations = { "Rural Area", "Suburban Edge", "Urban Infill", "Coastal Zone", "Mountain Base" };

string[] investmentTypes = { "Land Banking", "Development", "Agriculture", "Conservation" };

string[] soilTypes = { "Loam", "Clay", "Sand", "Silt", "Peat", "Chalky" };

decimal price = random.Next(50000, 500000) + (decimal)random.NextDouble() \* 2000;

decimal marketValue = price \* (decimal)(0.7 + random.NextDouble() \* 0.6); // 70% to 130% of price

bool infrastructureAccess = random.Next(2) == 0;

return new LandPlot(names[random.Next(names.Length)], Math.Round(price, 2), locations[random.Next(locations.Length)], Math.Round(marketValue, 2), investmentTypes[random.Next(investmentTypes.Length)], soilTypes[random.Next(soilTypes.Length)], infrastructureAccess);

}

// --- Manual Creation Methods (Copied and slightly improved input prompts) ---

static Product CreateManualProduct()

{

Console.Write("Enter Product Name: ");

string name = Console.ReadLine();

Console.Write("Enter Product Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine()); // Throws FormatException on bad input

return new Product(name, price); // Constructor validates price > 0

}

static RealEstate CreateManualRealEstate()

{

Console.Write("Enter RealEstate Name: ");

string name = Console.ReadLine();

Console.Write("Enter RealEstate Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Size (> 0): ");

double size = double.Parse(Console.ReadLine());

Console.Write("Enter Type (e.g., Residential, Commercial): ");

string type = Console.ReadLine();

return new RealEstate(name, price, location, size, type); // Constructor validates price/size

}

static RealEstateInvestment CreateManualRealEstateInvestment()

{

Console.Write("Enter Investment Name: ");

string name = Console.ReadLine();

Console.Write("Enter Investment Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Market Value (> 0): ");

decimal marketValue = decimal.Parse(Console.ReadLine());

Console.Write("Enter Investment Type (e.g., REIT, Direct Property): ");

string investmentType = Console.ReadLine();

return new RealEstateInvestment(name, price, location, marketValue, investmentType); // Constructor validates price/marketValue

}

static Apartment CreateManualApartment()

{

Console.Write("Enter Apartment Name: ");

string name = Console.ReadLine();

Console.Write("Enter Apartment Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Size (> 0): ");

double size = double.Parse(Console.ReadLine());

Console.Write("Enter Type (e.g., Condo, Co-op): ");

string type = Console.ReadLine();

Console.Write("Enter Floor Number (> 0): ");

int floorNumber = int.Parse(Console.ReadLine());

Console.Write("Enter HOA Fees (>= 0): ");

decimal hoaFees = decimal.Parse(Console.ReadLine());

return new Apartment(name, price, location, size, type, floorNumber, hoaFees); // Constructor validates

}

static House CreateManualHouse()

{

Console.Write("Enter House Name: ");

string name = Console.ReadLine();

Console.Write("Enter House Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Size (> 0): ");

double size = double.Parse(Console.ReadLine());

Console.Write("Enter Type (e.g., Single-family): ");

string type = Console.ReadLine();

Console.Write("Enter Garden Size (>= 0): ");

double gardenSize = double.Parse(Console.ReadLine());

Console.Write("Has Pool (true/false): ");

bool pool = bool.Parse(Console.ReadLine());

return new House(name, price, location, size, type, gardenSize, pool); // Constructor validates

}

static Hotel CreateManualHotel()

{

Console.Write("Enter Hotel Name: ");

string name = Console.ReadLine();

Console.Write("Enter Hotel Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Market Value (> 0): ");

decimal marketValue = decimal.Parse(Console.ReadLine());

Console.Write("Enter Investment Type: ");

string investmentType = Console.ReadLine();

Console.Write("Enter Number of Rooms (> 0): ");

int rooms = int.Parse(Console.ReadLine());

Console.Write("Enter Star Rating (1-5): ");

int starRating = int.Parse(Console.ReadLine());

return new Hotel(name, price, location, marketValue, investmentType, rooms, starRating); // Constructor validates

}

static LandPlot CreateManualLandPlot()

{

Console.Write("Enter LandPlot Name: ");

string name = Console.ReadLine();

Console.Write("Enter LandPlot Price (> 0): ");

decimal price = decimal.Parse(Console.ReadLine());

Console.Write("Enter Location: ");

string location = Console.ReadLine();

Console.Write("Enter Market Value (> 0): ");

decimal marketValue = decimal.Parse(Console.ReadLine());

Console.Write("Enter Investment Type: ");

string investmentType = Console.ReadLine();

Console.Write("Enter Soil Type (e.g., Loam, Clay): ");

string soilType = Console.ReadLine();

Console.Write("Has Infrastructure Access (true/false): ");

bool infrastructureAccess = bool.Parse(Console.ReadLine());

return new LandPlot(name, price, location, marketValue, investmentType, soilType, infrastructureAccess); // Constructor validates

}

}