+Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Інформатики та інтелектуальної власності"

**Звіт**

**Лабораторна робота 5**

з дисципліни "Кросплатформне програмування"

Виконав: студент групи КН-1224С

Мульков М. В.

Перевірив:

Івашко А.В.

Харків 2025

Зміст

[Завдання 3](#_Toc194089735)

[Реалізація коду 4](#_Toc194089736)

[Алгоритм коду 4](#_Toc194089737)

[Тестування коду 7](#_Toc194089738)

[Висновок 9](#_Toc194089739)

[Додаток А 10](#_Toc194089740)

**Робота з індексаторами у класі-контейнері**

**Мета роботи**: Розширити функціонал створеного у попередній лабораторній роботі класу Container шляхом додавання індексаторів, які б дозволяли отримувати доступ до об'єктів контейнера за порядковим номером додавання, за ім'ям товару та за його вартістю.

# Завдання

В рамках даної роботи необхідно:

Розширити функціонал створеного у попередній лабораторній роботі класу Container шляхом додавання індексаторів, які б повертали об'єкти:

За порядковим номером додавання елемента до контейнера.

За ім'ям товару елемента (Name).

За вартістю товару (Price).

# Реалізація коду

## Алгоритм коду

Повний код програми відображено у додатку А.

Для реалізації доступу до елементів контейнера за різними критеріями за допомогою синтаксису індексаторів, клас Container було модифіковано наступним чином:

1. Додавання відстеження порядку вставки:

Додано приватний масив insertionOrder типу int[], паралельний до основного масиву items. Додано приватний лічильник nextInsertionId типу int, який інкрементується при кожному додаванні елемента.

Метод Add модифіковано таким чином, щоб при додаванні нового об'єкта в items[count], відповідний унікальний ідентифікатор порядку вставки (nextInsertionId) зберігався у insertionOrder[count]. При розширенні масивів items та insertionOrder розширюються синхронно.

2. Реалізація індексатора за порядковим номером (int):

Створено індексатор public object this[int i]. У get-аксесорі виконується пошук у масиві insertionOrder індексу j, для якого insertionOrder[j] == i (де i – це шуканий порядковий номер вставки).

Якщо такий індекс j знайдено, повертається елемент items[j]. Якщо порядковий номер i не знайдено (або він виходить за межі діапазону), генерується виняток IndexOutOfRangeException.

3. Реалізація індексаторів за ім'ям (string) та ціною (decimal):

Створено індексатори public object[] this[string i] та public object[] this[decimal i]. Вони повертають масив object[], оскільки ім'я або ціна не обов'язково є унікальними.

Для реалізації пошуку за параметром створено універсальний приватний метод GetItemsByParameter<T>(string param, T value). Цей метод ітерує по масиву items (до count), і для кожного елемента за допомогою рефлексії (через інший допоміжний метод GetPropertyValue<T>) отримує значення властивості з ім'ям param.

Якщо отримане значення властивості дорівнює шуканому value, об'єкт додається до результуючого масиву.Індексатори за string та decimal просто викликають GetItemsByParameter, передаючи відповідно "Name" або "Price" як ім'я параметра.

4. Модифікація методів RemoveById та Sort:

Метод RemoveById змінено так, щоб при зсуві елементів у масиві items аналогічний зсув відбувався і в масиві insertionOrder, зберігаючи відповідність між елементом та його порядковим номером вставки відносно інших елементів, що залишилися.

Метод Sort модифіковано для синхронного сортування обох масивів (items та insertionOrder). Коли два елементи в items міняються місцями на основі порівняння їх цін, відповідні їм значення в insertionOrder також міняються місцями.

5. Оновлення ToString() в ієрархії класів:

Методи ToString() у похідних класах (RealEstate, Apartment тощо) були оновлені для включення виклику base.ToString(), що дозволяє виводити інформацію з базових класів, роблячи вивід більш повним та структурованим.

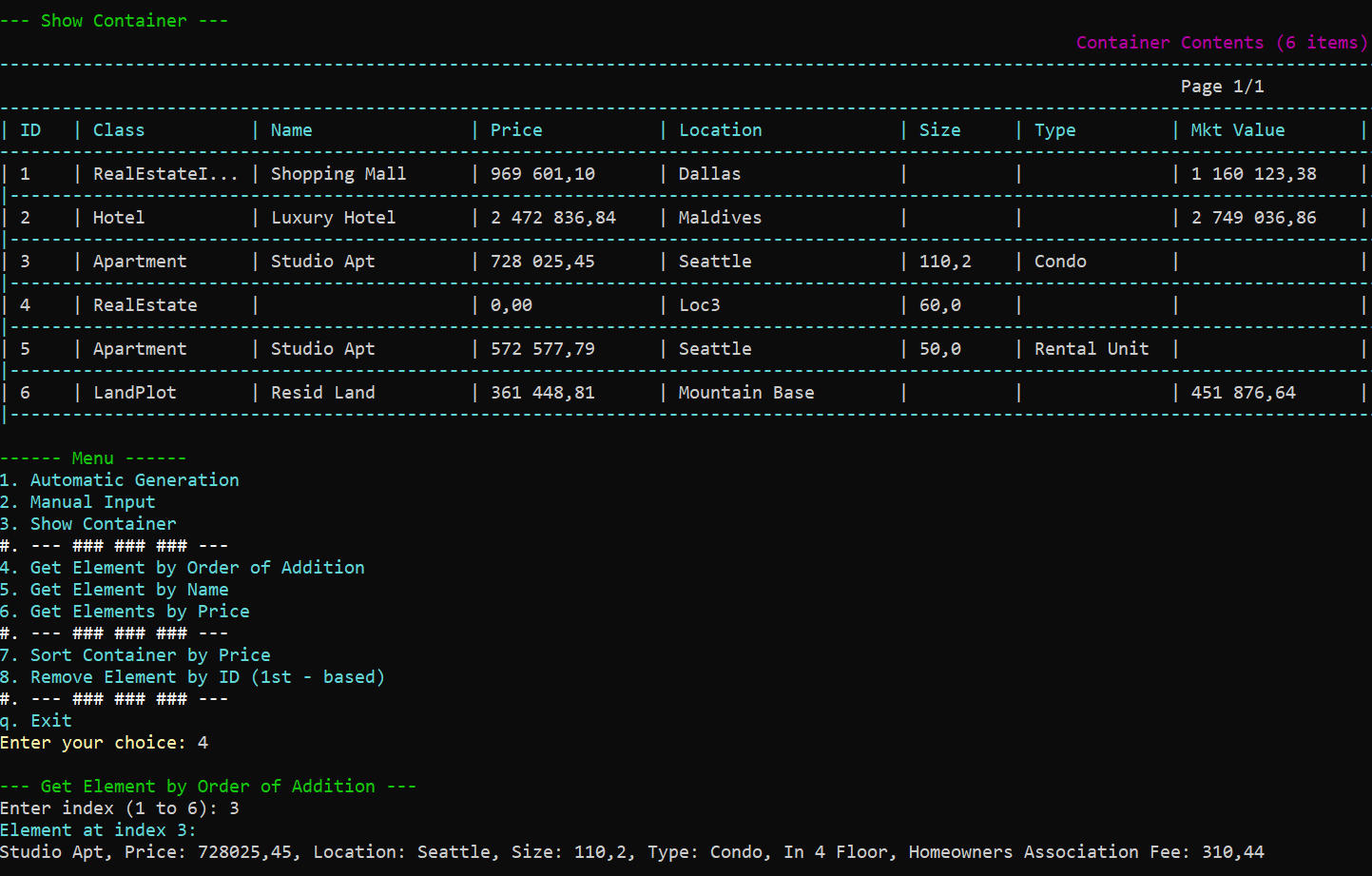
6. Оновлення консольної програми тестування:

Головне меню програми доповнено пунктами для тестування нових індексаторів (отримання за порядковим номером, за ім'ям, за ціною).Реалізовано відповідні функції (GetElementByOrderOfAddition, GetElementByName, GetElementsByPrice), які запитують у користувача необхідний ключ (індекс, ім'я або ціну) та виводять результат роботи індексатора.

Додано метод DemonstrateIndexers для автоматичної демонстрації роботи індексаторів після генерації випадкових даних.Оновлено обробку винятків для коректного перехоплення IndexOutOfRangeException, що може генеруватися індексаторами.

## Тестування коду

Тестування індексатора за порядковим номером зображено на рис. 1.



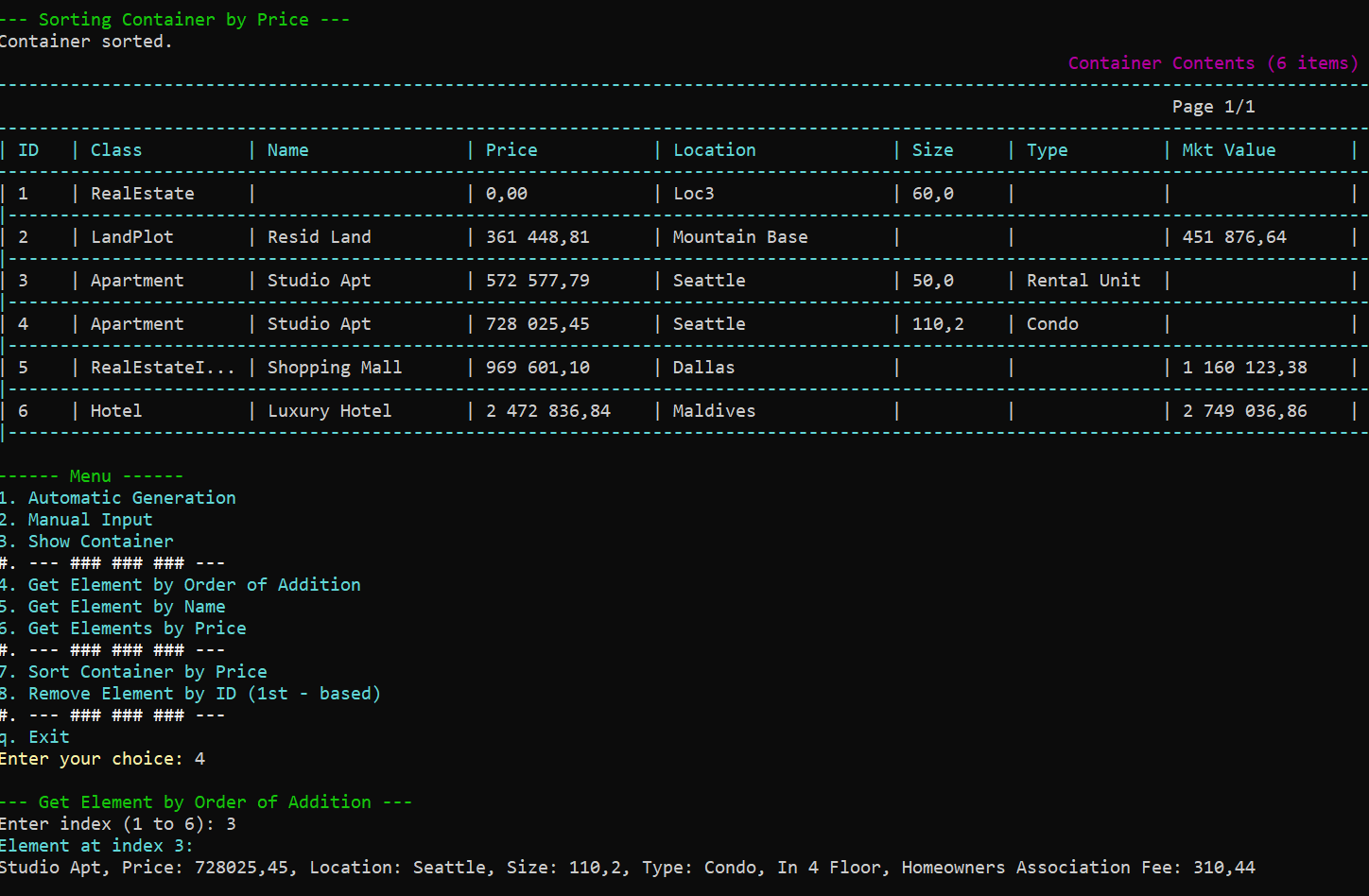


Рис. 1. Тестування індексатора за ім'ям

Тестування індексатора за ім'ям відображено на рис. 2.

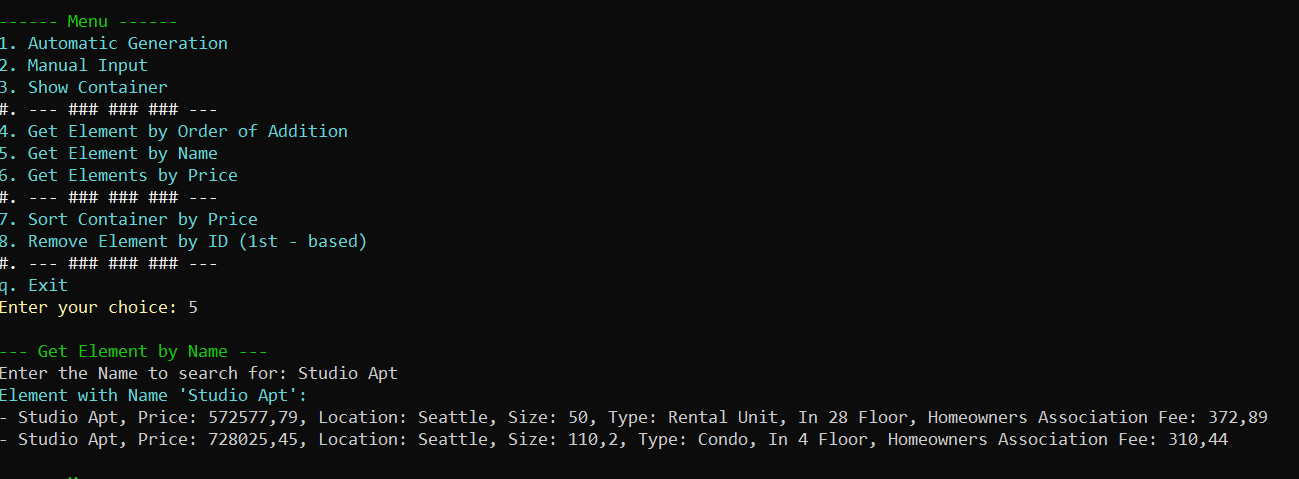


Рис. 2. Тестування індексатора за ім'ям

Тестування індексатора за ціною зображено на рис. 3.

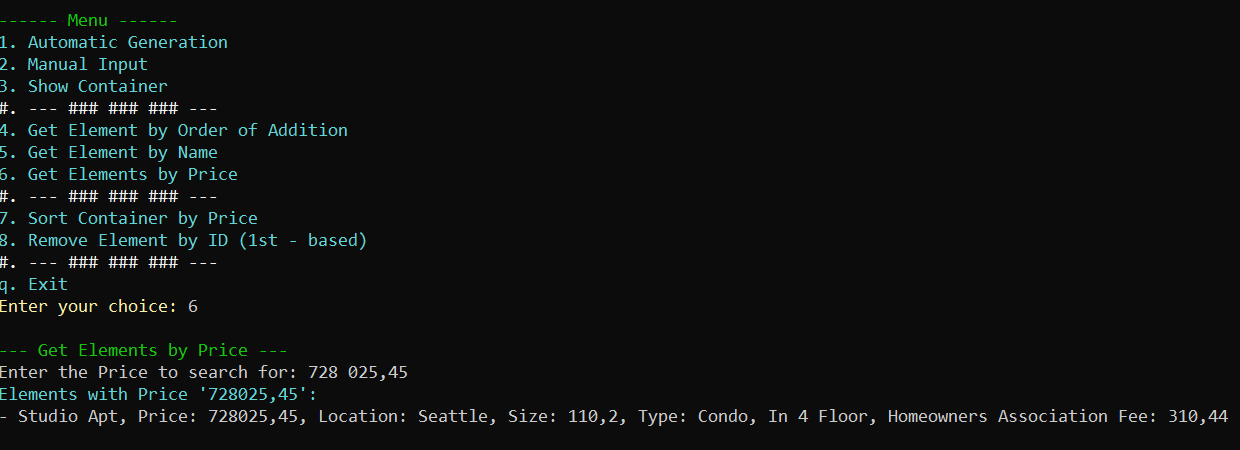


Рис. 3. Тестування індексатора за ціною

# Висновок

У ході виконання даної лабораторної роботи було успішно розширено функціонал класу Container шляхом додавання трьох типів індексаторів. Це дозволило реалізувати зручний та інтуїтивно зрозумілий синтаксис для доступу до елементів контейнера: за порядковим номером їх додавання, за ім'ям товару та за ціною.

Для реалізації індексатора за порядковим номером було введено додатковий масив та лічильник, а також модифіковано методи для підтримки цілісності даних про порядок вставки. Для реалізації індексаторів за ім'ям та ціною було використано рефлексію, що дозволило створити універсальний механізм пошуку за значенням властивості.

Оновлення методів у класах ієрархії покращило повноту виведення інформації про об'єкти. Модифікація консольної програми тестування дозволила наочно продемонструвати та перевірити працездатність реалізованих індексаторів.

# Додаток А

using System.Reflection;

namespace lb\_5;

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 $"{base.ToString()}, Location: {Location}, Size: {Size}, Type: {Type}";

}

}

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 $"{base.ToString()}, Location: {Location}, Market 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 $"{base.ToString()}, 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 $"{base.ToString()}, 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 $"{base.ToString()}, 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 $"{base.ToString()}, 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[] insertionOrder;

private int count;

private int size;

private int nextInsertionId;

public Container()

{

items = new Object[1];

insertionOrder = new int[1];

count = 0;

size = 1;

nextInsertionId = 0;

}

public void Add(object \_newObject)

{

if (count == size)

{

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

int[] newInsertionOrder = new int[size \* 2];

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

{

newArray[i] = items[i];

newInsertionOrder[i] = insertionOrder[i];

}

items = newArray;

insertionOrder = newInsertionOrder;

size \*= 2;

}

items[count] = \_newObject;

insertionOrder[count] = nextInsertionId++;

count++;

}

public object RemoveById(int \_index)

{

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

throw new IndexOutOfRangeException();

object deletedObject = items[\_index];

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

{

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

insertionOrder[i] = insertionOrder[i + 1];

}

items[count - 1] = null;

insertionOrder[count - 1] = 0;

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]);

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

}

public Object[] GetItemsByParameter<T>(string param, T i)

{

Object[] \_items = new Object[count];

int index = 0;

foreach (var item in items)

{

if (item != null)

{

var value = GetPropertyValue<T>(item, param);

if (value != null && value.Equals(i))

{

\_items[index] = item;

index++;

}

}

}

return index == 0 ? default : \_items;

}

public Object this[int i]

{

get

{

if (i < 0) throw new IndexOutOfRangeException();

if (i > nextInsertionId) throw new IndexOutOfRangeException($"There is no entry number {i}");

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

if (insertionOrder[j] == i)

return items[j];

return null;

}

}

public Object[] this[string i] => GetItemsByParameter("Name", i);

public Object[] this[decimal i] => GetItemsByParameter("Price", i);

}

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. Get Element by Order of Addition");

Console.WriteLine("5. Get Element by Name");

Console.WriteLine("6. Get Elements by Price");

Console.ForegroundColor = ConsoleColor.White;

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

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("7. Sort Container by Price");

Console.WriteLine("8. Remove Element by ID (1st - based)");

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();

DemonstrateIndexers(container, random);

}

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":

GetElementByOrderOfAddition(container);

break;

case "5":

GetElementByName(container);

break;

case "6":

GetElementsByPrice(container);

break;

case "7":

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 "8":

RemoveElementByIndex(container);

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();

}

}

}

// --- Indexer Interaction Methods ---

static void GetElementByOrderOfAddition(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Get Element by Order of Addition ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.WriteLine("Container is empty. Cannot get element by id.");

return;

}

Console.Write($"Enter index (1 to {container.GetCount()}): ");

if (int.TryParse(Console.ReadLine(), out int index))

{

try

{

object item = container[index - 1];

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($"Element at index {index}:");

Console.ResetColor();

Console.WriteLine(item?.ToString() ?? "Item not found (should not happen with valid index).");

}

catch (IndexOutOfRangeException)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error: Index {index} is out of range.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid input. Please enter a valid integer index.");

Console.ResetColor();

}

}

static void GetElementByName(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Get Element by Name ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.WriteLine("Container is empty. Cannot get element by name.");

return;

}

Console.Write("Enter the Name to search for: ");

string name = Console.ReadLine();

if (string.IsNullOrWhiteSpace(name))

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid input. Name cannot be empty.");

Console.ResetColor();

return;

}

Object[] items = container[name];

if (items != null)

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($"Element with Name '{name}':");

Console.ResetColor();

Console.ResetColor();

int foundCount = 0;

foreach (var item in items)

{

if (item != null)

{

Console.WriteLine($"- {item.ToString()}");

foundCount++;

}

}

if (foundCount == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"No elements found with Price '{name}'.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"No element found with Name '{name}'.");

Console.ResetColor();

}

}

static void GetElementsByPrice(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Get Elements by Price ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.WriteLine("Container is empty. Cannot get elements by price.");

return;

}

Console.Write("Enter the Price to search for: ");

if (decimal.TryParse(Console.ReadLine(), out decimal price))

{

Object[] items = container[price];

if (items != null)

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($"Elements with Price '{price}':");

Console.ResetColor();

int foundCount = 0;

foreach (var item in items)

{

if (item != null)

{

Console.WriteLine($"- {item.ToString()}");

foundCount++;

}

}

if (foundCount == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"No elements found with Price '{price}'.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"No elements found with Price '{price}'.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid input. Please enter a valid decimal price.");

Console.ResetColor();

}

}

static void RemoveElementByIndex(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Remove Element by 1-based ID ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

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

return;

}

Console.Write($"Enter element ID to remove (1 to {container.GetCount()}): ");

if (int.TryParse(Console.ReadLine(), out int id) && id >= 1 && id <= container.GetCount())

{

int index = id - 1;

try

{

object deletedItem = container.RemoveById(index);

Console.ForegroundColor = ConsoleColor.DarkCyan;

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

Console.ResetColor();

}

catch (IndexOutOfRangeException)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error: ID {id} is invalid.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Invalid input. Please enter a valid ID between 1 and {container.GetCount()}.");

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

}

static void DemonstrateIndexers(Container container, Random random)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("\n--- Demonstrating Indexer Usage ---");

Console.ResetColor();

int currentCount = container.GetCount();

if (currentCount == 0)

{

Console.WriteLine("Container is empty, cannot demonstrate indexers.");

return;

}

int demoIndex = random.Next(currentCount);

try

{

object itemByIndex = container[demoIndex];

Console.WriteLine($"1. Using int indexer container[{demoIndex}]:");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Found: {itemByIndex?.ToString() ?? "N/A"}");

Console.ResetColor();

}

catch (Exception ex) // Should not happen if index is valid

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($" Error getting item by index {demoIndex}: {ex.Message}");

Console.ResetColor();

}

string demoName = null;

object sourceItemForName = null;

for (int attempt = 0; attempt < Math.Min(5, currentCount); ++attempt)

{

int nameSearchIndex = random.Next(currentCount);

sourceItemForName = container[nameSearchIndex];

demoName = GetPropertyValue<string>(sourceItemForName, "Name");

if (!string.IsNullOrEmpty(demoName)) break;

}

Console.WriteLine($"\n2. Using string indexer container[\"{demoName ?? "N/A"}\"]:");

if (!string.IsNullOrEmpty(demoName))

{

try

{

object itemByName = container[demoName];

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Found: {itemByName?.ToString() ?? "Not Found"}");

if (itemByName != sourceItemForName)

Console.WriteLine($" (Note: Found item might differ from the source if names are duplicated)");

Console.ResetColor();

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($" Error getting item by name '{demoName}': {ex.Message}");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine(" Could not find an item with a non-empty name in random sampling to demonstrate.");

Console.ResetColor();

}

decimal demoPrice = -1m;

for (int attempt = 0; attempt < Math.Min(5, currentCount); ++attempt)

{

int priceSearchIndex = random.Next(currentCount);

object sourceItemForPrice = container[priceSearchIndex];

demoPrice = GetPropertyValue<decimal>(sourceItemForPrice, "Price");

if (demoPrice > 0) break; // Found a valid price

else demoPrice = -1m; // Reset if price was 0 or invalid

}

Console.WriteLine($"\n3. Using decimal indexer container[{demoPrice}]:");

if (demoPrice > 0)

{

try

{

Object[] itemsByPrice = container[demoPrice];

if (itemsByPrice != null)

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Found items with price {demoPrice}:");

int foundCount = 0;

foreach (var item in itemsByPrice)

{

if (item != null)

{

Console.WriteLine($" - {item.ToString()}");

foundCount++;

}

}

if (foundCount == 0) // Should not happen if itemsByPrice != null

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" No items found for price {demoPrice} (indexer returned non-null but empty?).");

}

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" No items found for price {demoPrice}.");

Console.ResetColor();

}

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($" Error getting items by price {demoPrice}: {ex.Message}");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine(" Could not find an item with a positive price in random sampling to demonstrate.");

Console.ResetColor();

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("--- End Indexer Demonstration ---");

Console.ResetColor();

}

// --- 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)

{

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() ?? "";

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

}

}