Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Інформатики та інтелектуальної власності"

**Звіт**

**Лабораторна робота 7**

з дисципліни "Кросплатформне програмування"

Виконав: студент групи КН-1224С

Мульков М. В.

Перевірив:

Івашко А.В.

Харків 2025

Зміст

[Завдання 3](#_Toc195219833)

[Реалізація коду 4](#_Toc195219834)

[Алгоритм коду 4](#_Toc195219835)

[Тестування коду 6](#_Toc195219836)

[Висновок 8](#_Toc195219837)

[Додаток А 9](#_Toc195219838)

**Робота з інтерфейсами у .NET Framework**

**Мета роботи**: Дослідити використання інтерфейсів у C# для визначення контрактів класів, модифікувати клас-контейнер для зберігання об'єктів, що реалізують конкретний інтерфейс.

# Завдання

В рамках даної роботи необхідно:

Створити інтерфейс IName, який успадковується від System.IComparable.

В інтерфейсі IName описати вимогу наявності рядкової властивості Name (з гетером та сетером).

Модифікувати клас Product та, відповідно, всю ієрархію товарів для реалізації інтерфейсу IName. Реалізувати метод CompareTo, необхідний для IComparable.

Переробити клас Container таким чином, щоб він зберігав об'єкти, які реалізують інтерфейс IName (замість object).

У класі Container залишити лише індексатори за порядковим номером додавання об'єктів та за ім'ям товару (Name). Видалити інші індексаторы.

# Реалізація коду

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення інтерфейсу IName

Створено файл IName.cs, де оголошено публічний інтерфейс IName.Вказано, що IName успадковується від стандартного інтерфейсу System.IComparable.В інтерфейсі оголошено сигнатуру властивості Name типу string з методами доступу get та set.

namespace lb\_7.Interfaces

{

public interface IName : IComparable

{

string Name { get; set; }

}

}

2. Реалізація IName у класі Product

Клас Product (базовий для всієї ієрархії) модифіковано для реалізації інтерфейсу IName. Оскільки клас Product вже мав публічну властивість string Name { get; set; }, це задовольнило вимогу інтерфейсу щодо властивості Name.

Реалізовано метод CompareTo(object obj), який вимагається інтерфейсом IComparable (успадкованим через IName). Метод порівнює поточний об'єкт з іншим об'єктом obj за значенням їх властивостей Name, використовуючи StringComparer.OrdinalIgnoreCase для порівняння без урахування регістру. Це дозволяє сортувати об'єкти за ім'ям.

using lb\_7.Interfaces;

namespace lb\_7.Classes

{

class Product : IName

{

public string Name { get; set; }

public decimal Price { get; set; }

public int CompareTo(object obj)

{

if (obj == null) return 1;

if (obj is not IName otherProduct)

{

throw new ArgumentException($"Object must be type {nameof(IName)}");

}

return StringComparer.OrdinalIgnoreCase.Compare(this.Name, otherProduct.Name);

}

}

}

3. Модифікація класу Container для роботи з IName

Тип внутрішнього масиву items змінено з object[] на IName?[]. Параметр методу Add змінено на тип IName.Типи повернення методів GetItems, GetItemsByParameter, GetInstanceByInsertionId та індексаторів оновлено до IName або IName[] відповідно. Це забезпечує типізацію контейнера – він може зберігати лише об'єкти, що реалізують IName.

class Container

{

private IName?[] items;

…

public Container()

{

items = new IName[1];

…

}

public void Add(IName \_newObject)

{

if (count == size)

{

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

…

}

4. Оновлення індексаторів у Container

Індексатор за порядковим номером (public IName? this[int id]) залишено. Він повертає об'єкт типу IName? за його унікальним ID вставки. Додано set-аксесор для можливості заміни об'єкта за його ID вставки.

Індексатор за ім'ям (public IName[] this[string i]) залишено. Він використовує метод GetItemsByParameter для пошуку всіх об'єктів типу IName з відповідним ім'ям і повертає масив IName[].

Індексатор за ціною (public IName[] this[decimal i]) було видалено (закоментовано), відповідно до завдання.

5. Модифікація консольної програми

Оновлено меню програми: видалено пункт для отримання елементів за ціною (пункт 6 у коді закоментовано).

Додано нові пункти меню (6 та 7) для тестування функціоналу зміни елементів за порядковим номером або за ім'ям (використовуючи set-аксесор індексатора та рефлексію у методі ModifyProperty).

Функції взаємодії з користувачем (GetElementBy..., ChangeItemBy..., ManualInput тощо) оновлено для коректної роботи з об'єктами типу IName.

У методі AutomaticGeneration та методах ручного створення об'єктів тепер створюються та додаються об'єкти, що гарантовано реалізують IName.

## Тестування коду

Тестування додавання згенерованих та введених вручну об'єктів з інтерфейсом IName відображено на рис. 1 та рис. 2, відображення контейнера на рис. 3.

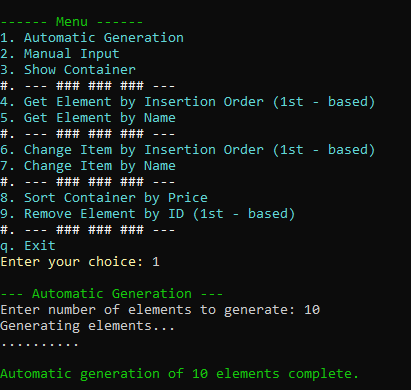


Рис. 1. Тестування додавання згенерованих об'єктів

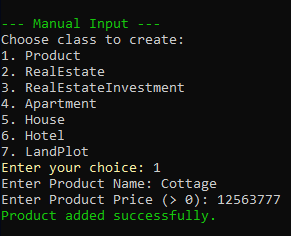


Рис. 2. Тестування введених вручну об'єкту

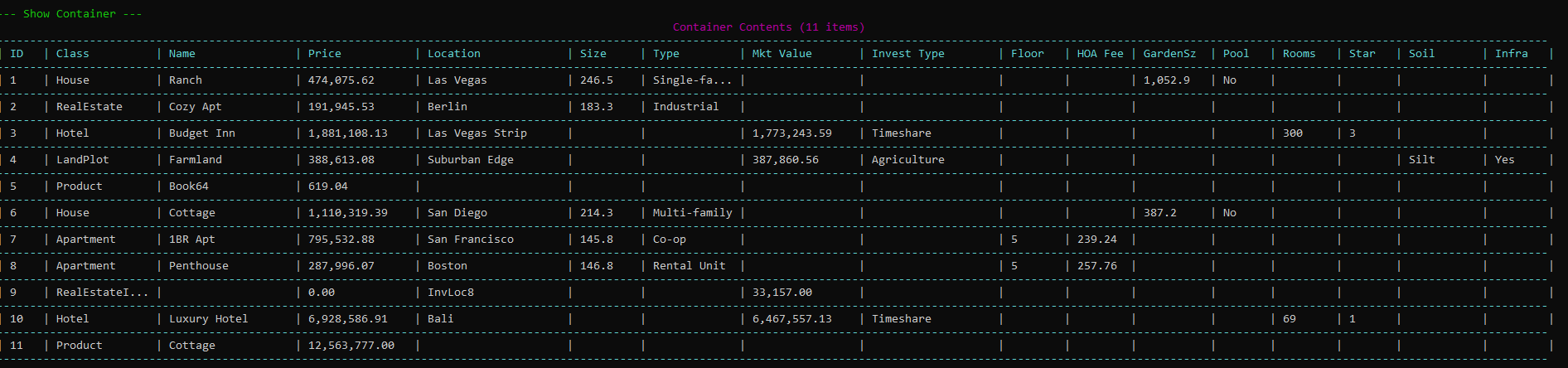


Рис. 3. Відображення контейнера

# Висновок

У ході виконання лабораторної роботи було успішно застосовано інтерфейси для визначення загального контракту IName для класів ієрархії товарів. Реалізація інтерфейсу IComparable у базовому класі Product дозволила забезпечити можливість порівняння та, як наслідок, сортування об'єктів ієрархії за ім'ям.

Клас Container було ефективно модифіковано для роботи з об'єктами, що реалізують інтерфейс IName, замість нетипізованого object. Це підвищило безпеку типів та надійність коду, оскільки контейнер тепер гарантовано містить об'єкти з властивістю Name та можливістю порівняння.

Індексатори класу Container було оновлено відповідно до завдання: залишено індексатори за порядковим номером вставки та за ім'ям, індексатор за ціною видалено. Було також додано функціонал для зміни об'єктів у контейнері через індексатор за ID вставки.

Робота дозволила отримати практичний досвід у використанні інтерфейсів для реалізації поліморфізму та визначення контрактів, застосуванні IComparable для сортування об'єктів, а також у модифікації існуючих класів для роботи з інтерфейсами замість базових типів чи object, що є важливим кроком до написання більш гнучкого, типізованого та масштабованого коду.

# Додаток А

namespace lb\_7.Interfaces

{

public interface IName : IComparable

{

string Name { get; set; }

}

}

using lb\_7.Interfaces;

namespace lb\_7.Classes

{

class Product : IName

{

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}";

}

public int CompareTo(object obj)

{

if (obj == null) return 1;

if (obj is not IName otherProduct)

{

throw new ArgumentException($"Object must be type {nameof(IName)}");

}

return StringComparer.OrdinalIgnoreCase.Compare(this.Name, otherProduct.Name);

}

}

}

namespace lb\_7.Classes

{

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}";

}

}

}

using lb\_7.Interfaces;

namespace lb\_7.Classes

{

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}";

}

}

}

namespace lb\_7.Classes

{

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";

}

}

}

namespace lb\_7.Classes

{

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";

}

}

}

namespace lb\_7.Classes

{

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}";

}

}

}

namespace lb\_7.Classes

{

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}";

}

}

}

using lb\_7.Interfaces;

using System.Reflection;

namespace lb\_7

{

class Container

{

private IName?[] items;

private int[] insertionOrder;

private int count;

private int size;

private int nextInsertionId;

public Container()

{

items = new IName[1];

insertionOrder = new int[1];

count = 0;

size = 1;

nextInsertionId = 0;

}

public void Add(IName \_newObject)

{

if (count == size)

{

IName[] newArray = new IName[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 override string ToString()

{

string res = "";

foreach (var item in items)

{

if (item is null)

continue;

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

}

return res;

}

public IName[] GetItems()

{

return items;

}

public int GetCount()

{

return count;

}

public int GetInsertionId()

{

return nextInsertionId;

}

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

{

IName[] \_items = new IName[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 IName? GetInstanceByInsertionId(int id)

{

if (id < 0 | id > nextInsertionId) throw new IndexOutOfRangeException($"There is no entry number {id}");

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

{

if (insertionOrder[j] == id)

{

return items[j];

}

}

return null;

}

public IName? this[int id] // Insertion order indexer

{

get => GetInstanceByInsertionId(id);

set

{

if (value == null) throw new ArgumentNullException(nameof(value));

IName \_item = GetInstanceByInsertionId(id);

if (\_item != null)

{

\_item = value;

}

throw new IndexOutOfRangeException("Can not find element by this insertion index");

}

}

public IName[] this[string i] // Name indexer

{

get => GetItemsByParameter("Name", i);

}

//public IName[] this[decimal i] // Price indexer

//{

// get => GetItemsByParameter("Price", i);

//}

}

}

using System.ComponentModel;

using System.Globalization;

using System.Reflection;

using lb\_7.Classes;

using lb\_7.Interfaces;

namespace lb\_7;

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 Insertion Order (1st - based)");

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("6. Change Item by Insertion Order (1st - based)");

Console.WriteLine("7. Change Item by Name");

Console.ForegroundColor = ConsoleColor.White;

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

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("8. Sort Container by Price");

Console.WriteLine("9. 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 "6":

ChangeItemByInsertionOrder(container);

break;

case "7":

ChangeItemByName(container);

break;

case "8":

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

if (container.GetCount() > 0)

{

container.Sort();

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("Container sorted.");

Console.ResetColor();

ShowContainer(container);

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

break;

case "9":

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/Validation 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: Index out of range. {ex.Message}");

Console.ResetColor();

}

catch (ArgumentException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

catch (TargetInvocationException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"\nError during operation: {ex.InnerException?.Message ?? ex.Message}");

Console.ResetColor();

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"\nAn unexpected error occurred: {ex.GetType().Name} - {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 Current Position (Table View) ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter position (1 to {container.GetInsertionId()}): ");

Console.ResetColor();

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

{

try

{

object item = container[index - 1];

if (item == null)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error: Item at position {index} is not found or was removed.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nElement at position {index}:");

Console.ResetColor();

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

WriteDataRow(index, item);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

catch (IndexOutOfRangeException)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

static void GetElementByName(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Get Elements by Name (Table View) ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

string name = Console.ReadLine();

if (string.IsNullOrWhiteSpace(name))

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

return;

}

IName[] itemsFound = container[name];

List<IName> validItems = new List<IName>();

if (itemsFound != null)

{

validItems.AddRange(itemsFound.Where(item => item != null));

}

if (validItems.Count > 0)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nFound {validItems.Count} element(s) with Name '{name}':");

Console.ResetColor();

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

object[] allItems = container.GetItems();

int currentCount = container.GetCount();

foreach (var foundItem in validItems)

{

int originalPosition = -1;

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

{

if (object.ReferenceEquals(allItems[i], foundItem))

{

originalPosition = i + 1;

break;

}

}

if (originalPosition != -1)

{

WriteDataRow(originalPosition, foundItem);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"| Warning: Could not find original position for item {foundItem.ToString()}".PadRight(tableWidth - 1) + "|");

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

}

//static void GetElementsByPrice(Container container)

//{

// Console.ForegroundColor = ConsoleColor.Green;

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

// Console.ResetColor();

// if (container.GetCount() == 0)

// {

// Console.ForegroundColor = ConsoleColor.Yellow;

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

// Console.ResetColor();

// return;

// }

// Console.ForegroundColor = ConsoleColor.Yellow;

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

// Console.ResetColor();

// if (decimal.TryParse(Console.ReadLine(), NumberStyles.Any, CultureInfo.InvariantCulture, out decimal price))

// {

// IName[] itemsFound = container.GetItemsByParameter("Price", price);

// List<IName> validItems = new List<IName>();

// if (itemsFound != null)

// {

// validItems.AddRange(itemsFound.Where(item => item != null));

// }

// if (validItems.Count > 0)

// {

// Console.ForegroundColor = ConsoleColor.Green;

// Console.WriteLine($"\nFound {validItems.Count} element(s) with Price '{price:N2}':");

// Console.ResetColor();

// int tableWidth = CalculateTableWidth();

// Console.ForegroundColor = ConsoleColor.Cyan;

// DrawHorizontalLine(tableWidth);

// WriteHeaderRow();

// DrawHorizontalLine(tableWidth);

// Console.ResetColor();

// object[] allItems = container.GetItems();

// int currentCount = container.GetCount();

// foreach (var foundItem in validItems)

// {

// int originalPosition = -1;

// for (int i = 0; i < currentCount; i++)

// {

// if (object.ReferenceEquals(allItems[i], foundItem))

// {

// originalPosition = i + 1;

// break;

// }

// }

// if (originalPosition != -1)

// {

// WriteDataRow(originalPosition, foundItem);

// Console.ForegroundColor = ConsoleColor.Cyan;

// DrawHorizontalLine(tableWidth);

// Console.ResetColor();

// }

// else

// {

// Console.ForegroundColor = ConsoleColor.Yellow;

// Console.WriteLine($"| Warning: Could not find original position for item {foundItem.ToString()}".PadRight(tableWidth - 1) + "|");

// DrawHorizontalLine(tableWidth);

// Console.ResetColor();

// }

// }

// }

// else

// {

// Console.ForegroundColor = ConsoleColor.Yellow;

// Console.WriteLine($"No elements found with Price '{price:N2}'.");

// Console.ResetColor();

// }

// }

// else

// {

// Console.ForegroundColor = ConsoleColor.Red;

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

// Console.ResetColor();

// }

//}

static void ChangeItemByInsertionOrder(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Change Item by Current Position ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Container is empty. Cannot change elements.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter item insortion position to modify (1 to {container.GetInsertionId()}): ");

Console.ResetColor();

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

{

try

{

object itemToModify = container[index - 1];

if (itemToModify == null)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error: Item at position {index} is not found or was removed.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nCurrent item details:");

Console.ResetColor();

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

WriteDataRow(index, itemToModify);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

Console.ResetColor();

ModifyProperty(itemToModify, index);

}

catch (IndexOutOfRangeException)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

static void ChangeItemByName(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\n--- Change Item by Name ---");

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Container is empty. Cannot change elements.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("Enter the Name of the item(s) to modify: ");

Console.ResetColor();

string name = Console.ReadLine();

if (string.IsNullOrWhiteSpace(name))

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

return;

}

IName[] itemsFound = container[name];

List<IName> validItems = new List<IName>();

if (itemsFound != null)

{

validItems.AddRange(itemsFound.Where(item => item != null));

}

if (validItems.Count == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($"No valid elements found matching Name '{name}'.");

Console.ResetColor();

return;

}

IName itemToModify;

int originalPosition = -1;

if (validItems.Count == 1)

{

itemToModify = validItems[0];

object[] allItems = container.GetItems();

for (int j = 0; j < container.GetCount(); j++) { if (object.ReferenceEquals(allItems[j], itemToModify)) { originalPosition = j + 1; break; } }

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nFound one item (Position: {originalPosition}):");

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nFound {validItems.Count} items with Name '{name}'. Please choose which one to modify:");

Console.ResetColor();

object[] allItems = container.GetItems();

Dictionary<int, int> choiceToPositionMap = new Dictionary<int, int>();

for (int i = 0; i < validItems.Count; i++)

{

int currentOriginalPosition = -1;

for (int j = 0; j < container.GetCount(); j++) { if (object.ReferenceEquals(allItems[j], validItems[i])) { currentOriginalPosition = j + 1; break; } }

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($"{i + 1}. (Position: {currentOriginalPosition}) {validItems[i].ToString()}");

Console.ResetColor();

if (currentOriginalPosition != -1)

{

choiceToPositionMap[i + 1] = currentOriginalPosition;

}

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter choice (1 to {validItems.Count}): ");

Console.ResetColor();

if (int.TryParse(Console.ReadLine(), out int choice) && choice >= 1 && choice <= validItems.Count)

{

itemToModify = validItems[choice - 1];

choiceToPositionMap.TryGetValue(choice, out originalPosition);

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Invalid choice.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nSelected item (Position: {originalPosition}):");

Console.ResetColor();

}

if (originalPosition != -1 && itemToModify != null)

{

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

WriteDataRow(originalPosition, itemToModify);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("\nDetails of selected item:");

Console.WriteLine(itemToModify?.ToString() ?? "N/A");

Console.ResetColor();

}

ModifyProperty(itemToModify, originalPosition);

}

static void ModifyProperty(object itemToModify, int itemPosition) // Added itemPosition for context

{

if (itemToModify == null) throw new ArgumentNullException(nameof(itemToModify));

var properties = itemToModify.GetType()

.GetProperties(BindingFlags.Public | BindingFlags.Instance)

.Where(p => p.CanWrite)

.ToList();

if (properties.Count == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("This object has no writable properties.");

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nChoose property to modify:");

Console.ResetColor();

for (int i = 0; i < properties.Count; i++)

{

object currentValue = "?";

try { currentValue = properties[i].GetValue(itemToModify); } catch { /\* Ignore errors here \*/ }

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($"{i + 1}. {properties[i].Name} (Type: {properties[i].PropertyType.Name}, Current: '{currentValue ?? "null"}')");

Console.ResetColor();

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter choice (1 to {properties.Count}): ");

Console.ResetColor();

if (int.TryParse(Console.ReadLine(), out int propChoice) && propChoice >= 1 && propChoice <= properties.Count)

{

PropertyInfo selectedProperty = properties[propChoice - 1];

Type propertyType = selectedProperty.PropertyType;

Type underlyingType = Nullable.GetUnderlyingType(propertyType);

bool isNullable = underlyingType != null;

Type targetType = underlyingType ?? propertyType;

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter new value for {selectedProperty.Name} (Type: {targetType.Name}{(isNullable ? ", or empty for null" : "")}): ");

Console.ResetColor();

string newValueString = Console.ReadLine();

object convertedValue = null;

if (isNullable && string.IsNullOrEmpty(newValueString))

{

convertedValue = null;

}

else

{

try

{

TypeConverter converter = TypeDescriptor.GetConverter(targetType);

if (converter != null && converter.CanConvertFrom(typeof(string)))

{

CultureInfo culture = CultureInfo.InvariantCulture;

if (targetType == typeof(decimal) || targetType == typeof(double) || targetType == typeof(float))

{

convertedValue = converter.ConvertFromString(null, culture, newValueString);

}

else if (targetType == typeof(bool))

{

string lowerVal = newValueString.ToLowerInvariant().Trim();

if (lowerVal == "true" || lowerVal == "1" || lowerVal == "yes" || lowerVal == "y")

convertedValue = true;

else if (lowerVal == "false" || lowerVal == "0" || lowerVal == "no" || lowerVal == "n")

convertedValue = false;

else

throw new FormatException($"Cannot convert '{newValueString}' to Boolean.");

}

else

{

convertedValue = converter.ConvertFromString(null, culture, newValueString);

}

}

else

{

convertedValue = Convert.ChangeType(newValueString, targetType, CultureInfo.InvariantCulture);

}

}

catch (Exception ex) when (ex is FormatException || ex is InvalidCastException || ex is NotSupportedException || ex is ArgumentException)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Conversion Error: Could not convert '{newValueString}' to type {targetType.Name}. {ex.Message}");

Console.ResetColor();

return;

}

}

try

{

selectedProperty.SetValue(itemToModify, convertedValue);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nProperty '{selectedProperty.Name}' updated successfully.");

Console.WriteLine("New item details:");

Console.ResetColor();

if (itemPosition != -1)

{

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

WriteDataRow(itemPosition, itemToModify);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine(itemToModify?.ToString() ?? "N/A");

Console.ResetColor();

}

}

catch (TargetInvocationException tie)

{

throw tie.InnerException ?? tie;

}

catch (ArgumentException argEx)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error setting property: {argEx.Message}");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

static void RemoveElementByIndex(Container container)

{

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

if (container.GetCount() == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

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

{

int index = id - 1;

try

{

object itemToRemove = container.GetItems()[index];

object deletedItem = container.RemoveById(index);

Console.ForegroundColor = ConsoleColor.DarkCyan;

Console.WriteLine($"\nElement (Position: {id}) was removed:");

Console.WriteLine(itemToRemove?.ToString() ?? "Removed item details unavailable.");

Console.ResetColor();

}

catch (IndexOutOfRangeException)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Error: Position {id} is invalid or item already removed.");

Console.ResetColor();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"Invalid input. Please enter a valid position 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))

{

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:

switch (random.Next(1, 7))

{

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

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

case 3: container.Add(new Apartment(random.Next(1, 10), random.Next(50, 300))); break;

case 4: container.Add(new House(random.Next(100, 500), random.Next(2) == 0)); break;

case 5: container.Add(new Hotel(random.Next(20, 100), random.Next(1, 6))); break;

case 6: container.Add(new LandPlot($"Soil{i}", random.Next(2) == 0)); break;

}

break;

}

Console.Write(".");

}

Console.WriteLine();

}

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.GetItems()[demoIndex];

Console.WriteLine($"1. Accessing item at current position [{demoIndex + 1}]:");

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.ResetColor();

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($" Error getting item by position {demoIndex + 1}: {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.GetItems()[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

{

IName[] itemsByName = container[demoName];

if (itemsByName != null)

{

var validItems = itemsByName.Where(it => it != null).ToList();

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Found {validItems.Count} item(s):");

foreach (var item in validItems)

{

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

}

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" Indexer returned null for name '{demoName}'.");

Console.ResetColor();

}

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($" Error getting item(s) 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();

}

// 3. Demonstrate getting by Price (using GetItemsByParameter)

//decimal demoPrice = -1m;

//object sourceItemForPrice = null;

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

//{

// int priceSearchIndex = random.Next(currentCount);

// sourceItemForPrice = container.GetItems()[priceSearchIndex];

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

// if (demoPrice > 0) break;

// else demoPrice = -1m;

//}

//Console.WriteLine($"\n3. Using GetItemsByParameter(\"Price\", {demoPrice:N2}):");

//if (demoPrice > 0)

//{

// try

// {

// IName[] itemsByPrice = container.GetItemsByParameter("Price", demoPrice);

// if (itemsByPrice != null)

// {

// var validItems = itemsByPrice.Where(it => it != null).ToList();

// Console.ForegroundColor = ConsoleColor.Cyan;

// Console.WriteLine($" Found {validItems.Count} item(s):");

// foreach (var item in validItems)

// {

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

// }

// Console.ResetColor();

// }

// else

// {

// Console.ForegroundColor = ConsoleColor.Yellow;

// Console.WriteLine($" No items found for price {demoPrice:N2}.");

// Console.ResetColor();

// }

// }

// catch (Exception ex)

// {

// Console.ForegroundColor = ConsoleColor.Red;

// Console.WriteLine($" Error getting items by price {demoPrice:N2}: {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();

IName 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;

}

if (newItem != null)

{

container.Add(newItem);

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

}

}

catch (ValueLessThanZero ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

catch (FormatException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

}

}

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;

}

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

Console.ResetColor();

Object[] items = container.GetItems();

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

{

var item = items[i];

if (item == null) continue;

WriteDataRow(i + 1, item);

Console.ForegroundColor = ConsoleColor.Cyan;

DrawHorizontalLine(tableWidth);

}

}

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

const int idWidth = 4;

const int classWidth = 14;

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;

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

+ (17 \* 3);

}

static void WriteHeaderRow()

{

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.Write($"| {"Class".PadRight(classWidth)} ");

Console.Write($"| {"Name".PadRight(nameWidth)} ");

Console.Write($"| {"Price".PadRight(priceWidth)} ");

Console.Write($"| {"Location".PadRight(locationWidth)} ");

Console.Write($"| {"Size".PadRight(sizeWidth)} ");

Console.Write($"| {"Type".PadRight(typeWidth)} ");

Console.Write($"| {"Mkt Value".PadRight(marketValueWidth)} ");

Console.Write($"| {"Invest Type".PadRight(investmentTypeWidth)} ");

Console.Write($"| {"Floor".PadRight(floorWidth)} ");

Console.Write($"| {"HOA Fee".PadRight(hoaWidth)} ");

Console.Write($"| {"GardenSz".PadRight(gardenWidth)} ");

Console.Write($"| {"Pool".PadRight(poolWidth)} ");

Console.Write($"| {"Rooms".PadRight(roomsWidth)} ");

Console.Write($"| {"Star".PadRight(starWidth)} ");

Console.Write($"| {"Soil".PadRight(soilWidth)} ");

Console.Write($"| {"Infra".PadRight(infraWidth)} ");

Console.WriteLine("|");

Console.ResetColor();

}

static void WriteDataRow(int id, object item)

{

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

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

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

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

Type itemType = item.GetType();

string name = GetPropertyValue<string>(item, "Name");

string formattedPrice = FormatDecimal(GetPropertyValue<decimal?>(item, "Price"));

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

string formattedSize = FormatDouble(GetPropertyValue<double?>(item, "Size"));

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

string formattedMarketValue = FormatDecimal(GetPropertyValue<decimal?>(item, "MarketValue"));

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

string formattedFloorNumber = FormatInt(GetPropertyValue<int?>(item, "FloorNumber"));

string formattedHoaFees = FormatDecimal(GetPropertyValue<decimal?>(item, "HOAFees"));

string formattedGardenSize = FormatDouble(GetPropertyValue<double?>(item, "GardenSize"));

string formattedPool = FormatBool(GetPropertyValue<bool?>(item, "Pool"));

string formattedRooms = FormatInt(GetPropertyValue<int?>(item, "Rooms"));

string formattedStarRating = FormatInt(GetPropertyValue<int?>(item, "StarRating"));

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

string formattedInfrastructureAccess = FormatBool(GetPropertyValue<bool?>(item, "InfrastructureAccess"));

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

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

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

Console.Write($"| {Truncate(formattedPrice, priceWidth).PadRight(priceWidth)} ");

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

Console.Write($"| {Truncate(formattedSize, sizeWidth).PadRight(sizeWidth)} ");

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

Console.Write($"| {Truncate(formattedMarketValue, marketValueWidth).PadRight(marketValueWidth)} ");

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

Console.Write($"| {Truncate(formattedFloorNumber, floorWidth).PadRight(floorWidth)} ");

Console.Write($"| {Truncate(formattedHoaFees, hoaWidth).PadRight(hoaWidth)} ");

Console.Write($"| {Truncate(formattedGardenSize, gardenWidth).PadRight(gardenWidth)} ");

Console.Write($"| {Truncate(formattedPool, poolWidth).PadRight(poolWidth)} ");

Console.Write($"| {Truncate(formattedRooms, roomsWidth).PadRight(roomsWidth)} ");

Console.Write($"| {Truncate(formattedStarRating, starWidth).PadRight(starWidth)} ");

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

Console.Write($"| {Truncate(formattedInfrastructureAccess, infraWidth).PadRight(infraWidth)} ");

Console.WriteLine("|");

}

static void DrawHorizontalLine(int tableWidth)

{

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.ResetColor();

}

static string CenterString(string s, int width)

{

if (string.IsNullOrEmpty(s)) return new string(' ', width);

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

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 "";

if (maxLength <= 3) return value.Length <= maxLength ? value : new string('.', maxLength);

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, BindingFlags.Public | BindingFlags.Instance);

if (property != null)

{

if (typeof(T).IsAssignableFrom(property.PropertyType))

{

try

{

object value = property.GetValue(item);

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

{

return default(T);

}

return (T)value;

}

catch

{

return default;

}

}

else if (Nullable.GetUnderlyingType(typeof(T)) == property.PropertyType)

{

try

{

object value = property.GetValue(item);

return (T)value;

}

catch

{

return default;

}

}

}

return default;

}

// --- Random Generators ---

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);

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;

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.Max(0, 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);

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

int starRating = random.Next(1, 6);

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);

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 ---

static Product CreateManualProduct()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

return new Product(name, price);

}

static RealEstate CreateManualRealEstate()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

double size = double.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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

string type = Console.ReadLine();

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

}

static RealEstateInvestment CreateManualRealEstateInvestment()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

decimal marketValue = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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

string investmentType = Console.ReadLine();

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

}

static Apartment CreateManualApartment()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

double size = double.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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(), CultureInfo.InvariantCulture);

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

}

static House CreateManualHouse()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

double size = double.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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

string type = Console.ReadLine();

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

double gardenSize = double.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Has Pool (true/false/yes/no/1/0): ");

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

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

}

static Hotel CreateManualHotel()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

decimal marketValue = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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);

}

static LandPlot CreateManualLandPlot()

{

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

string name = Console.ReadLine();

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

decimal price = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

Console.Write("Enter Location: ");

string location = Console.ReadLine();

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

decimal marketValue = decimal.Parse(Console.ReadLine(), CultureInfo.InvariantCulture);

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/yes/no/1/0): ");

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

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

}

}