Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Інформатики та інтелектуальної власності"

Звіт

Лабораторна робота 13

з дисципліни "Кросплатформне програмування"

Виконав: студент групи КН-1224С

Мульков М. В.

Перевірив:

Івашко А.В.

Харків 2025

Зміст

[Завдання 3](#_Toc198013883)

[Реалізація коду 4](#_Toc198013884)

[Алгоритм коду 4](#_Toc198013885)

[Тестування коду 6](#_Toc198013886)

[Висновок 10](#_Toc198013887)

[Додаток А 11](#_Toc198013888)

Використання подій для динамічного відстеження сумарної вартості у контейнерах у .NET Framework

Мета роботи: Розширити функціонал узагальнених класів-контейнерів для забезпечення можливості отримання сумарної вартості всіх товарів, що в них містяться, без необхідності перегляду кожного елемента. Реалізувати механізм, що динамічно оновлює сумарну вартість при додаванні, видаленні або зміні ціни окремого товару в контейнері, використовуючи події.

# Завдання

В рамках даної роботи необхідно:

1. На основі попередньої лабораторної роботи розширити функціонал узагальнених класів-контейнерів (Container<T> та ContainerLinkedList<T>).
2. Реалізувати для обох контейнерів властивість (або метод) для отримання сумарної вартості всіх товарів, що реалізують інтерфейс IPrice.
3. Забезпечити, щоб сумарна вартість у контейнері автоматично оновлювалася при:
   * Додаванні нового товару, що має ціну.
   * Видаленні товару, що мав ціну.
   * Зміні ціни існуючого товару в контейнері.
4. Для реалізації відстеження зміни ціни товару:
   * Додати подію PriceChanged до інтерфейсу IPrice (або до класів, що його реалізують).
   * У класі Product (та, відповідно, у його нащадках) реалізувати генерацію цієї події при зміні властивості Price.
   * У класах-контейнерах підписатися на подію PriceChanged для кожного доданого товару, що реалізує IPrice, та відписатися при його видаленні. Обробник події має коригувати сумарну вартість.
5. Модифікувати консольну програму тестування для демонстрації отримання сумарної вартості та перевірки її коректного оновлення при різних операціях з товарами.

# Реалізація коду

## Алгоритм коду

Повний код програми відображено у додатку А.

Модифікація інтерфейсу IPrice:

* До інтерфейсу IPrice додано оголошення події event EventHandler<decimal>? PriceChanged;. Параметр decimal у EventHandler призначений для передачі старої ціни товару, що дозволить контейнеру коректно розрахувати зміну загальної суми.

Реалізація події PriceChanged у класі Product:

* У класі Product властивість Price перетворено на повну властивість з приватним полем price.
* У set-аксесорі властивості Price додано логіку:
* Перевірка, чи нова ціна не є меншою або рівною нулю (генерується ValueLessThanZero, якщо так).
* Перевірка, чи дійсно змінилася ціна (this.price != value).
* Якщо ціна змінилася, зберігається стара ціна, оновлюється значення поля price, а потім генерується подія PriceChanged?.Invoke(this, oldPrice), передаючи об'єкт-відправник (this) та стару ціну.
* Реалізація відстеження сумарної вартості у Container<T> (на основі масиву):
* Додано приватне поле private decimal totalPrice; для зберігання сумарної вартості.
* Додано публічну властивість public decimal TotalPrice => totalPrice; лише для читання.
* Метод Add(T \_newObject):
  + Якщо доданий об'єкт \_newObject реалізує IPrice, його ціна додається до totalPrice.
  + Здійснюється підписка на подію \_newObject.PriceChanged методом-обробником HandlePriceChange.
* Метод RemoveById(int \_index):
  + Якщо видалений об'єкт deletedObject реалізує IPrice, його ціна віднімається від totalPrice.
  + Здійснюється відписка від події deletedObject.PriceChanged.
* Індексатор set ( this[int id] ):
  + При заміні існуючого об'єкта новим, якщо старий об'єкт реалізував IPrice, його ціна віднімається від totalPrice і відбувається відписка від його події PriceChanged.
  + Якщо новий об'єкт value реалізує IPrice, його ціна додається до totalPrice і відбувається підписка на його подію PriceChanged.
* Приватний метод HandlePriceChange(object? sender, decimal oldPrice):
  + Цей метод викликається при спрацьовуванні події PriceChanged у будь-якого об'єкта в контейнері.
  + Якщо sender є IPrice, то різниця між новою ціною (price.Price) та старою ціною (oldPrice) додається до totalPrice, таким чином коригуючи загальну суму.
* Метод PostDeserializeInitialize(): (важливо при десеріалізації)
  + Після десеріалізації контейнера цей метод викликається для повторної підписки на події PriceChanged для всіх елементів, що реалізують IPrice. Це необхідно, оскільки підписки на події не серіалізуються. Поле totalPrice при цьому не перераховується, оскільки передбачається, що ціни об'єктів вже коректні після десеріалізації, а totalPrice буде серіалізовано/десеріалізовано як частина стану контейнера (якщо це реалізовано в ContainerSerializer). Однак, у наданому коді ContainerSerializer не серіалізує totalPrice самого контейнера, тому PostDeserializeInitialize мав би також перерахувати totalPrice з нуля, або totalPrice мало б бути властивістю, що обчислюється при кожному запиті, якщо події не використовуються для цього після десеріалізації.
* Реалізація відстеження сумарної вартості у ContainerLinkedList<T>:
* Аналогічно до Container<T>, додано поле totalPrice та властивість TotalPrice.
* Методи AddFirst(T data), AddLast(T data), RemoveByIndex(int index) та set-аксесор індексатора this[int insertionIdValue] модифіковано для оновлення totalPrice та підписки/відписки від події PriceChanged об'єктів, що реалізують IPrice.
* Реалізовано приватний метод HandlePriceChange для обробки змін цін.
* Додано метод PostDeserializeInitialize() для відновлення підписок на події та, потенційно, перерахунку totalPrice після десеріалізації.
* Модифікація консольної програми (Program.cs):
* Додано новий пункт меню (18) "Show Total Price of Items in Active Container".
* Реалізовано метод-обробник HandleShowTotalPrice, який перевіряє наявність активного контейнера та виводить значення його властивості TotalPrice, відформатоване як валюта.
* При тестуванні зміни ціни об'єкта (через пункти меню 6 або 7) перевірялося, що totalPrice контейнера оновлюється коректно.

## Тестування коду

Тестування нової функціональності проводилося для обох типів контейнерів:

1. Початкова сумарна вартість: Після створення порожнього контейнера перевірялося, що TotalPrice дорівнює 0.
2. Оновлення при додаванні: До контейнера додавалися різні об'єкти (включаючи ті, що реалізують IPrice, та ті, що ні, якщо такі були б можливі). Перевірялося, що TotalPrice коректно збільшується на ціну кожного доданого товару з ціною.
3. Оновлення при видаленні: З контейнера видалялися об'єкти. Перевірялося, що TotalPrice коректно зменшується на ціну видаленого товару (якщо він мав ціну).
4. Оновлення при зміні ціни:
   * Обиравався існуючий товар у контейнері.
   * Його властивість Price змінювалася через відповідний пункт меню (який використовує рефлексію для встановлення нового значення властивості).
   * Перевірялося, що подія PriceChanged у товарі спрацювала, обробник у контейнері був викликаний, і TotalPrice контейнера коректно оновилася.
5. Отримання сумарної вартості: Багаторазово викликався пункт меню для відображення TotalPrice після різних операцій для перевірки актуальності значення.
6. Робота з різними типами контейнерів: Перевірялося, що логіка відстеження сумарної вартості працює однаково коректно як для Container<T>, так і для ContainerLinkedList<T>.
7. Десеріалізація: Після десеріалізації контейнера перевірялося, що TotalPrice відображає коректну суму (це залежить від того, чи серіалізується totalPrice самого контейнера, чи перераховується в PostDeserializeInitialize). І що подальші зміни цін об'єктів у відновленому контейнері також коректно оновлюють totalPrice.

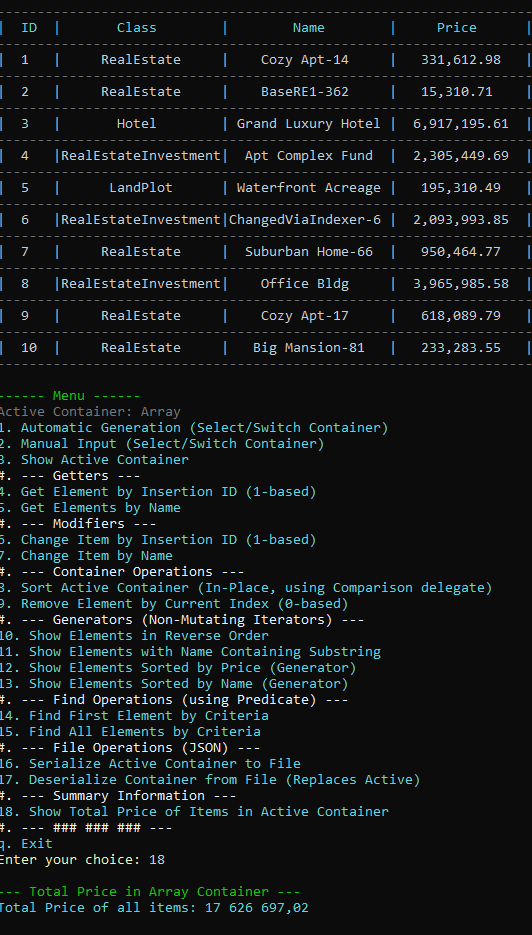


Рис. 1. Отримання загальної цiни

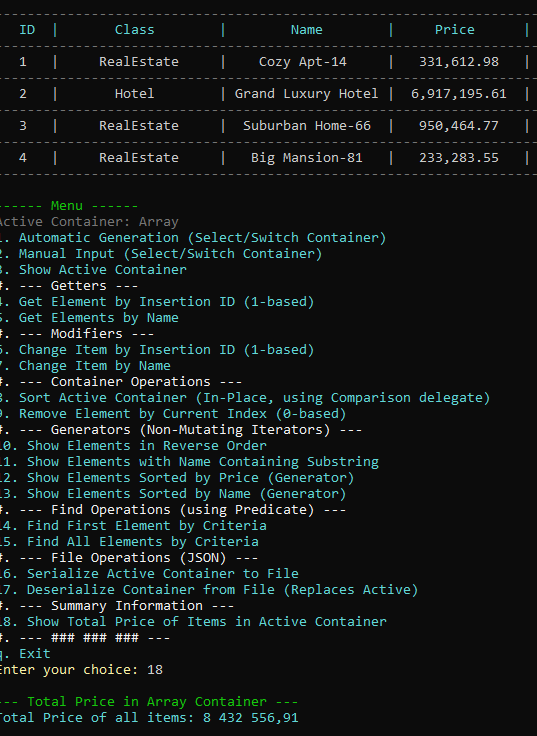


Рис. 2. Оновлення загальної цiни пiсля видалення товарiв

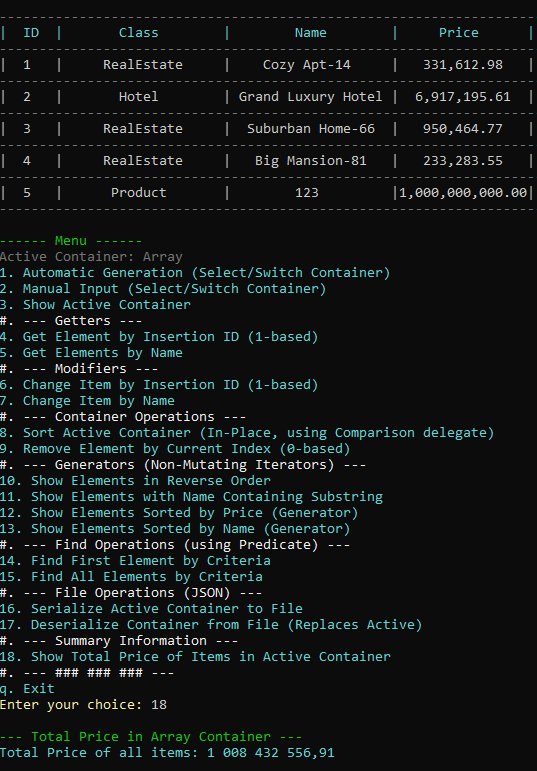


Рис. 3. Оновлення загальної цiни пiсля додавання товару

# Висновок

У ході лабораторної роботи було успішно реалізовано механізм динамічного відстеження сумарної вартості товарів в узагальнених контейнерах Container<T> та ContainerLinkedList<T>. Це досягнуто завдяки використанню подій: інтерфейс IPrice було розширено подією PriceChanged, яку генерує клас Product при зміні ціни. Контейнери підписуються на цю подію для об'єктів, що додаються, та відписуються при їх видаленні або заміні, а також оновлюють загальну суму при додаванні/видаленні.

Такий підхід дозволяє отримувати актуальну сумарну вартість у будь-який момент без необхідності перебору та підсумовування цін усіх елементів контейнера, що є ефективним з точки зору продуктивності, особливо для великих колекцій.

Реалізація PostDeserializeInitialize важлива для відновлення підписок на події після десеріалізації, оскільки самі підписки не серіалізуються. Консольна програма була доповнена для демонстрації та тестування нової функціональності.

Виконання роботи дозволило отримати практичний досвід у використанні подій для реалізації слабкозв'язаної взаємодії між об'єктами (товар та контейнер), що є важливим аспектом розробки гнучких та масштабованих систем.

# Додаток А

using System.Collections.Generic;

using lb\_8.Interfaces;

using System.Reflection;

namespace lb\_11

{

class Helper

{

public static V? GetPropertyValue<V>(object item, string propertyName)

{

if (item == null) return default;

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

if (property != null && property.CanRead)

{

return (V?)property.GetValue(item);

}

return default;

}

}

class Container<T> where T : class, IName

{

private T?[] items;

private int[] insertionOrder;

private int count;

private int size;

private int nextInsertionId;

public Container()

{

items = new T?[1];

insertionOrder = new int[1];

count = 0;

size = 1;

nextInsertionId = 0;

}

public void Add(T \_newObject)

{

if (count == size)

{

T?[] newArray = new T?[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 T? RemoveById(int \_index)

{

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

throw new IndexOutOfRangeException();

T? deletedObject = items[\_index]!;

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

{

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

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

}

items[count - 1] = default;

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 (Helper.GetPropertyValue<decimal>(items[j], "Price") > Helper.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();

}

}

public override string ToString()

{

string res = "";

foreach (var item in items)

{

if (item is null)

continue;

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

}

return res;

}

public T?[] GetItems()

{

return items;

}

public int GetCount()

{

return count;

}

public int GetInsertionId()

{

return nextInsertionId;

}

public int[] GetInsertionOrder()

{

return insertionOrder;

}

public bool IsEmpty(bool printMessage = true)

{

if (count == 0)

{

if (printMessage)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Container is empty");

Console.ResetColor();

}

return true;

}

return false;

}

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

{

var \_items = new T[count];

int index = 0;

foreach (var item in items)

{

if (item != null)

{

var value = Helper.GetPropertyValue<Y>(item, param);

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

{

\_items[index] = item;

index++;

}

}

}

return index == 0 ? default : \_items;

}

public T? 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 default;

}

// Insertion order indexer

public T? this[int id]

{

get => GetInstanceByInsertionId(id);

set

{

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

T? \_item = GetInstanceByInsertionId(id);

if (\_item != null)

{

\_item = value;

}

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

}

}

// Name indexer

public T[] this[string i]

{

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

}

//// Price indexer

//public T[] this[decimal i]

//{

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

//}

}

class ContainerLinkedList<T> where T : class, IName

{

public class Node<V>

{

public V Data { get; set; }

public Node<V> Next { get; set; }

public Node<V> Previous { get; set; }

public Node(V data)

{

Data = data;

Next = null;

Previous = null;

}

}

public ContainerLinkedList()

{

\_head = null;

Count = 0;

InsertionOrder = new List<int>();

NextInsertionId = 0;

}

private Node<T> \_head;

public Node<T> First => \_head;

public Node<T> Last => GetLastNode();

private int \_count;

public int Count

{

get

{

if (\_count < 0)

{

\_count = 0;

}

return \_count;

}

private set => \_count = value;

}

private int NextInsertionId;

private List<int> InsertionOrder;

public void AddFirst(T data)

{

Node<T> newNode = new Node<T>(data);

if (\_head != null)

{

newNode.Next = \_head;

\_head.Previous = newNode;

}

\_head = newNode;

Count++;

InsertionOrder.Add(NextInsertionId++);

}

public void AddLast(T data)

{

Node<T> newNode = new Node<T>(data);

if (\_head == null)

{

\_head = newNode;

}

else

{

Node<T> lastNode = GetLastNode();

lastNode.Next = newNode;

newNode.Previous = lastNode;

}

Count++;

InsertionOrder.Add(NextInsertionId++);

}

private Node<T> GetLastNode()

{

Node<T> node = \_head;

while (node.Next != null)

{

node = node.Next;

}

return node;

}

public T? RemoveByIndex(int index)

{

int count = 0;

var current = \_head;

while (current != null && count < index)

{

current = current.Next;

count++;

}

if (current == null) throw new ArgumentOutOfRangeException(nameof(index));

T? deletedItem = current.Data;

if (current.Previous != null)

current.Previous.Next = current.Next;

else

\_head = current.Next;

if (current.Next != null)

current.Next.Previous = current.Previous;

Count--;

InsertionOrder.RemoveAt(index);

return deletedItem;

}

public void Sort(string sortBy = "Price")

{

if (\_head == null) return;

List<T> list = new List<T>();

for (var node = \_head; node != null; node = node.Next)

{

list.Add(node.Data);

}

BinaryInsertionSort(list, sortBy);

var current = \_head;

foreach (var item in list)

{

current.Data = item;

current = current.Next;

}

}

private void BinaryInsertionSort(List<T> list, string propertyName)

{

for (int i = 1; i < list.Count; i++)

{

int currentInsertionValue = InsertionOrder[i];

T currentItem = list[i];

decimal currentValue = Helper.GetPropertyValue<decimal>(currentItem, propertyName);

int ins = BinarySearch(list, currentValue, propertyName, 0, i);

if (ins < i)

{

list.RemoveAt(i);

list.Insert(ins, currentItem);

InsertionOrder.RemoveAt(i);

InsertionOrder.Insert(ins, currentInsertionValue);

}

}

}

private int BinarySearch(List<T> list, decimal key, string propertyName, int low, int high)

{

while (low < high)

{

int mid = low + (high - low) / 2;

decimal midValue = Helper.GetPropertyValue<decimal>(list[mid], propertyName);

if (key < midValue)

high = mid;

else

low = mid + 1;

}

return low;

}

// Clear Container

public void Clear()

{

\_head = null;

Count = 0;

InsertionOrder.Clear();

NextInsertionId = 0;

}

public int GetNextInsertionId()

{

return NextInsertionId;

}

public List<int> GetInsertionOrder()

{

return InsertionOrder;

}

public override string ToString()

{

if (\_head is null) return "Container is empty.";

string res = string.Empty;

var current = \_head;

while (current != null)

{

res += current.Data?.ToString() + "\n";

current = current.Next;

}

return res;

}

private List<T> GetItemsByParameter<Y>(string parameter, Y i)

{

List<T> values = new List<T>();

var current = \_head;

while (current != null)

{

var propValue = Helper.GetPropertyValue<Y>(current.Data, parameter);

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

{

values.Add(current.Data);

}

current = current.Next;

}

return values.Count == 0 ? null : values;

}

// Insortion indexer

public T? this[int index]

{

get

{

var current = \_head;

int count = 0;

while (current != null)

{

if (InsertionOrder[count] == index)

return current.Data;

current = current.Next;

count++;

}

return null;

}

set

{

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

var current = \_head;

int count = 0;

while (current != null)

{

if (InsertionOrder[count] == index)

{

current.Data = value;

return;

}

current = current.Next;

count++;

}

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

}

}

// Name indexer

public List<T> this[string name]

{

get => GetItemsByParameter<string>("Name", name);

}

}

}

using lb\_8.Interfaces;

namespace lb\_8.Classes

{

class Product : IName, IName<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}";

}

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);

}

}

}

using System.ComponentModel;

using System.Globalization;

using System.Reflection;

using lb\_8.Classes;

using lb\_8.Interfaces;

namespace lb\_8;

enum ContainerType

{

None,

Array,

LinkedList

}

class Program

{

static Container<IName>? containerArray = null;

static ContainerLinkedList<IName>? containerList = null;

static ContainerType activeContainerType = ContainerType.None;

static Random random = new Random();

static void Main()

{

while (true)

{

PrintMenu();

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

try

{

switch (choice)

{

case "1": HandleContainerSelectionAndAction(HandleAutomaticGeneration); break;

case "2": HandleContainerSelectionAndAction(HandleManualInput); break;

case "3": HandleShowContainer(); break;

case "4": HandleGetElementByInsertionId(); break;

case "5": HandleGetElementByName(); break;

case "6": HandleChangeItemByInsertionId(); break;

case "7": HandleChangeItemByName(); break;

case "8": HandleSortContainer(); break;

case "9": HandleRemoveElementByIndex(); 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)

{

PrintErrorMessage($"Input/Validation Error: {ex.Message}");

}

catch (FormatException ex)

{

PrintErrorMessage($"Input Format Error: Invalid format entered. {ex.Message}");

}

catch (IndexOutOfRangeException ex)

{

PrintErrorMessage($"Error: Index out of range. {ex.Message}");

}

catch (KeyNotFoundException ex)

{

PrintErrorMessage($"Error: Key (e.g., Insertion ID) not found. {ex.Message}");

}

catch (ArgumentException ex)

{

PrintErrorMessage($"Argument Error: {ex.Message}");

}

catch (TargetInvocationException ex)

{

Exception inner = ex.InnerException ?? ex;

while (inner.InnerException != null) { inner = inner.InnerException; }

PrintErrorMessage($"Error during operation: {inner.GetType().Name} - {inner.Message}");

}

catch (Exception ex)

{

PrintErrorMessage($"An unexpected error occurred: {ex.GetType().Name} - {ex.Message}");

}

finally

{

Console.ResetColor();

}

}

}

// --- Menu Printing ---

static void PrintMenu()

{

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ForegroundColor = ConsoleColor.DarkGray;

Console.WriteLine($"Active Container: {activeContainerType}");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("1. Automatic Generation (Select/Switch Container)");

Console.WriteLine("2. Manual Input (Select/Switch Container)");

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

Console.ForegroundColor = ConsoleColor.White;

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

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("4. Get Element by Insertion ID (1-based)");

Console.WriteLine("5. Get Elements 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 ID (1-based)");

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

Console.ForegroundColor = ConsoleColor.White;

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

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.WriteLine("9. Remove Element by Current Index (0-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();

}

static void PrintErrorMessage(string message)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"\nERROR: {message}");

Console.ResetColor();

}

// --- Container Selection Logic ---

static void HandleContainerSelectionAndAction(Action actionToPerform)

{

ContainerType chosenType = AskContainerType();

if (chosenType == ContainerType.None)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Operation cancelled.");

Console.ResetColor();

return;

}

if (activeContainerType != chosenType || (activeContainerType == ContainerType.None))

{

bool switchConfirmed = true;

if (activeContainerType != ContainerType.None && activeContainerType != chosenType)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Switching to {chosenType} container will clear the current {activeContainerType} container. Continue? (y/n): ");

Console.ResetColor();

switchConfirmed = (Console.ReadLine()?.Trim().ToLower() == "y");

}

if (switchConfirmed)

{

Console.ForegroundColor = ConsoleColor.Magenta;

Console.WriteLine($"\nInitializing {chosenType} container...");

Console.ResetColor();

containerArray = null;

containerList = null;

activeContainerType = chosenType;

if (activeContainerType == ContainerType.Array)

{

containerArray = new Container<IName>();

}

else // LinkedList

{

containerList = new ContainerLinkedList<IName>();

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Switch cancelled. Keeping the current container.");

Console.ResetColor();

return;

}

}

else

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nContinuing with the active {activeContainerType} container.");

Console.ResetColor();

}

actionToPerform();

}

static ContainerType AskContainerType()

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("\nSelect Container Type:");

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("1. Array-based Container");

Console.WriteLine("2. Linked List Container");

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("Enter choice (1 or 2, or any other key to cancel): ");

Console.ResetColor();

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

return choice switch

{

"1" => ContainerType.Array,

"2" => ContainerType.LinkedList,

\_ => ContainerType.None,

};

}

// --- Action Handlers ---

static void HandleAutomaticGeneration()

{

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)

{

if (activeContainerType == ContainerType.Array)

{

AutomaticGenerationArray(containerArray!, random, count);

DemonstrateIndexersArray(containerArray!, random);

}

else // LinkedList

{

AutomaticGenerationList(containerList!, random, count);

DemonstrateIndexersList(containerList!, random);

}

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

}

else

{

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

}

}

static void HandleManualInput()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Manual Input for {activeContainerType} Container ---");

Console.ResetColor();

if (activeContainerType == ContainerType.Array)

{

ManualInputArray(containerArray!);

}

else // LinkedList

{

ManualInputList(containerList!);

}

}

static void HandleShowContainer()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Show {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out int currentCount))

{

// Message printed by IsContainerEmpty

return;

}

if (activeContainerType == ContainerType.Array)

{

ShowContainerArray(containerArray!, currentCount);

}

else // LinkedList

{

ShowContainerList(containerList!, currentCount);

}

}

// Still gets by Insertion ID, displays current index

static void HandleGetElementByInsertionId()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Get Element by Insertion ID from {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out \_)) return;

int maxId = GetNextInsertionId();

if (maxId == 0)

{

PrintErrorMessage("Container is empty, no IDs to get.");

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter insertion ID (1 to {maxId}): ");

Console.ResetColor();

if (int.TryParse(Console.ReadLine(), out int inputId) && inputId >= 1 && inputId <= maxId)

{

int insertionId = inputId - 1;

IName? item = null;

try

{

if (activeContainerType == ContainerType.Array)

{

item = containerArray![insertionId];

}

else // LinkedList

{

item = containerList![insertionId];

}

}

catch (IndexOutOfRangeException)

{

PrintErrorMessage($"Item with insertion ID {inputId} not found or invalid for container structure.");

return;

}

catch (Exception ex)

{

PrintErrorMessage($"Error accessing item with insertion ID {inputId}: {ex.Message}");

return;

}

if (item == null)

{

PrintErrorMessage($"Item with insertion ID {inputId} not found (possibly removed or ID never used/valid).");

return;

}

int currentIndex = FindIndexByReference(item);

if (currentIndex == -1)

{

PrintErrorMessage($"Found item by insertion ID {inputId}, but could not determine its current index for display.");

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nItem Details (Insertion ID: {inputId}, Current Index: {currentIndex}):");

Console.ResetColor();

DisplayItemTable(currentIndex + 1, item);

}

else

{

PrintErrorMessage($"Invalid input. Please enter a valid integer ID between 1 and {maxId}.");

}

}

static void HandleGetElementByName()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Get Elements by Name from {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out \_)) return;

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

string name = Console.ReadLine() ?? "";

if (string.IsNullOrWhiteSpace(name))

{

PrintErrorMessage("Invalid input. Name cannot be empty.");

return;

}

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

if (activeContainerType == ContainerType.Array)

{

IName[]? itemsFoundArray = containerArray![name];

if (itemsFoundArray != null)

{

itemsFoundList.AddRange(itemsFoundArray.Where(i => i != null)!);

}

}

else // LinkedList

{

List<IName>? itemsFoundLinkedList = containerList![name];

if (itemsFoundLinkedList != null)

{

itemsFoundList.AddRange(itemsFoundLinkedList);

}

}

if (itemsFoundList.Count > 0)

{

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

int tableWidth = CalculateTableWidth();

PrintTableHeader(tableWidth);

foreach (var foundItem in itemsFoundList)

{

int currentIndex = FindIndexByReference(foundItem);

if (currentIndex != -1)

{

WriteDataRowByDisplayId(currentIndex + 1, foundItem, tableWidth);

}

else

{

// Item was found by name indexer but couldn't be located by reference

Console.ForegroundColor = ConsoleColor.Yellow;

string itemStr = foundItem.ToString() ?? "N/A";

Console.WriteLine($"|{PadAndCenter($"Warning: Could not determine current index for item '{itemStr.Substring(0, Math.Min(20, itemStr.Length))}...'", tableWidth - 2)}|");

Console.ResetColor();

}

DrawHorizontalLine(tableWidth);

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

}

// Still changes by Insertion ID, displays current id

static void HandleChangeItemByInsertionId()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Change Item by Insertion ID in {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out \_)) return;

int maxId = GetNextInsertionId();

if (maxId == 0)

{

PrintErrorMessage("Container is empty, no IDs to modify.");

return;

}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter item insertion ID to modify (1 to {maxId}): ");

Console.ResetColor();

if (int.TryParse(Console.ReadLine(), out int inputId) && inputId >= 1 && inputId <= maxId)

{

int insertionId = inputId - 1;

IName? itemToModify = null;

try

{

if (activeContainerType == ContainerType.Array)

{

itemToModify = containerArray![insertionId];

}

else // LinkedList

{

itemToModify = containerList![insertionId];

}

}

catch (IndexOutOfRangeException)

{

PrintErrorMessage($"Item with insertion ID {inputId} not found or invalid for container structure.");

return;

}

catch (Exception ex)

{

PrintErrorMessage($"Error accessing item with insertion ID {inputId}: {ex.Message}");

return;

}

if (itemToModify == null)

{

PrintErrorMessage($"Item with insertion ID {inputId} not found (possibly removed or ID never used/valid).");

return;

}

int currentIndex = FindIndexByReference(itemToModify);

if (currentIndex == -1)

{

PrintErrorMessage($"Found item by insertion ID {inputId}, but could not determine its current index for display.");

return;

}

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

DisplayItemTable(currentIndex + 1, itemToModify);

ModifyProperty(itemToModify, insertionId);

}

else

{

PrintErrorMessage($"Invalid input. Please enter a valid integer ID between 1 and {maxId}.");

}

}

static void HandleChangeItemByName()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Change Item by Name in {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out \_)) 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))

{

PrintErrorMessage("Invalid input. Name cannot be empty.");

return;

}

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

if (activeContainerType == ContainerType.Array)

{

IName[]? itemsFoundArray = containerArray![name];

if (itemsFoundArray != null) validItems.AddRange(itemsFoundArray.Where(i => i != null)!);

}

else // LinkedList

{

List<IName>? itemsFoundList = containerList![name];

if (itemsFoundList != null) validItems.AddRange(itemsFoundList);

}

if (validItems.Count == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

return;

}

IName itemToModify;

int itemInsertionId = -1;

int currentDisplayIndex = -1;

if (validItems.Count == 1)

{

itemToModify = validItems[0];

itemInsertionId = GetInsertionIdForItem(itemToModify);

currentDisplayIndex = FindIndexByReference(itemToModify);

if (itemInsertionId == -1 || currentDisplayIndex == -1) { PrintErrorMessage("Could not find ID or index for the item."); return; }

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nFound one item (Current Index: {currentDisplayIndex + 1}, Insertion ID: {itemInsertionId + 1}):");

Console.ResetColor();

}

else // Multiple items found

{

Console.ForegroundColor = ConsoleColor.Green;

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

Console.ResetColor();

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

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

{

int currentItemIndex = FindIndexByReference(validItems[i]);

if (currentItemIndex != -1)

{

string itemInfo = validItems[i].ToString() ?? "N/A";

Console.WriteLine($"{i + 1}. (Index: {currentItemIndex + 1}) {itemInfo}");

choiceToCurrentIndexMap[i + 1] = currentItemIndex;

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

string itemStr = validItems[i].ToString() ?? "N/A";

Console.WriteLine($"{i + 1}. (Index: ???) Could not map item - {itemStr}");

Console.ResetColor();

}

}

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

&& choiceToCurrentIndexMap.TryGetValue(choice, out int chosenCurrentIndex))

{

itemToModify = GetItemByCurrentIndex(chosenCurrentIndex);

if (itemToModify == null) { PrintErrorMessage("Failed to re-acquire selected item by current index."); return; }

itemInsertionId = GetInsertionIdForItem(itemToModify);

currentDisplayIndex = chosenCurrentIndex;

if (itemInsertionId == -1) { PrintErrorMessage("Could not determine insertion ID for the chosen item."); return; }

}

else

{

PrintErrorMessage("Invalid choice or item mapping failed.");

return;

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nSelected item (Current Index: {currentDisplayIndex + 1}, Insertion ID: {itemInsertionId + 1}):");

Console.ResetColor();

}

// Modify the selected item

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

{

DisplayItemTable(currentDisplayIndex + 1, itemToModify);

ModifyProperty(itemToModify, itemInsertionId);

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("\nCould not reliably identify the selected item or its index. Modification cancelled.");

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

Console.ResetColor();

}

}

static void HandleSortContainer()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Sorting {activeContainerType} Container by Price ---");

Console.ResetColor();

if (IsContainerEmpty(out int currentCount)) return;

if (currentCount > 0)

{

if (activeContainerType == ContainerType.Array)

{

containerArray!.Sort();

}

else // LinkedList

{

containerList!.Sort("Price");

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("Container sorted.");

Console.ResetColor();

HandleShowContainer();

}

}

static void HandleRemoveElementByIndex()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\n--- Remove Element by Current Index from {activeContainerType} Container ---");

Console.ResetColor();

if (IsContainerEmpty(out int currentCount)) return;

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write($"Enter element index to remove (0 to {currentCount - 1}): ");

Console.ResetColor();

if (int.TryParse(Console.ReadLine(), out int index) && index >= 0 && index < currentCount)

{

IName? removedItem = null;

int removedItemInsertionId = -1;

try

{

IName? itemToRemove = GetItemByCurrentIndex(index);

if (itemToRemove != null)

{

removedItemInsertionId = GetInsertionIdForItem(itemToRemove);

}

else

{

PrintErrorMessage($"Could not retrieve item at index {index} before removal.");

}

if (activeContainerType == ContainerType.Array)

{

removedItem = containerArray!.RemoveById(index);

}

else // LinkedList

{

removedItem = containerList!.RemoveByIndex(index);

}

if (removedItem != null)

{

Console.ForegroundColor = ConsoleColor.DarkCyan;

string idString = removedItemInsertionId != -1 ? $"(original Insertion ID: {removedItemInsertionId + 1})" : "(original Insertion ID unknown)";

Console.WriteLine($"\nElement at index {index} {idString} was removed:");

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

Console.ResetColor();

}

else

{

PrintErrorMessage($"Error: Failed to remove item at index {index}. Item might have been null unexpectedly or removal failed.");

}

}

catch (Exception ex)

{

PrintErrorMessage($"Error during removal at index {index}: {ex.Message}");

}

}

else

{

PrintErrorMessage($"Invalid input. Please enter a valid index between 0 and {currentCount - 1}.");

}

}

// --- Indexer Interaction Methods ---

// Array Container Indexer Demonstration

static void DemonstrateIndexersArray(Container<IName> container, Random random)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

if (container.IsEmpty(false)) return;

int currentCount = container.GetCount();

int nextId = container.GetInsertionId();

// 1. Demonstrate Insertion ID Indexer (Get)

if (nextId > 0)

{

int demoInsertionId = random.Next(nextId);

Console.WriteLine($"1. Accessing item by random insertion ID [{demoInsertionId + 1}]:");

try

{

IName? itemById = container[demoInsertionId];

if (itemById != null)

{

int currentIndex = FindIndexByReference(itemById);

string indexStr = currentIndex != -1 ? $"(Current Index: {currentIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" Item with insertion ID {demoInsertionId + 1} not found (likely removed or ID never used/valid).");

Console.ResetColor();

}

}

catch (Exception ex)

{

PrintErrorMessage($" Error getting item by insertion ID {demoInsertionId + 1}: {ex.Message}");

}

}

else

{

Console.WriteLine("1. No items added yet, cannot demonstrate insertion ID indexer.");

}

// 2. Demonstrate Name Indexer (Get)

string? demoName = FindDemoName(container.GetItems(), container.GetCount(), random);

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

if (!string.IsNullOrWhiteSpace(demoName))

{

try

{

IName[]? itemsByName = container[demoName];

if (itemsByName != null && itemsByName.Any(i => i != null))

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Found {itemsByName.Count(i => i != null)} item(s):");

foreach (var item in itemsByName.Where(i => i != null))

{

int currentIndex = FindIndexByReference(item!);

string indexStr = currentIndex != -1 ? $"(Index: {currentIndex + 1})" : "";

Console.WriteLine($" - {indexStr} {item!.ToString() ?? "N/A"}");

}

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" No items found for name '{demoName}'.");

Console.ResetColor();

}

}

catch (Exception ex)

{

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

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

// 3. Demonstrate Insertion ID Indexer (Set)

int validDemoId = FindValidInsertionId(container);

if (validDemoId != -1)

{

Console.WriteLine($"\n3. Attempting to change item with insertion ID [{validDemoId + 1}] using property modification:");

IName? itemToModify = container[validDemoId];

if (itemToModify != null)

{

int currentIndex = FindIndexByReference(itemToModify);

string indexStr = currentIndex != -1 ? $"(Current Index: {currentIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Magenta;

Console.WriteLine($" Original Item {indexStr}: '{itemToModify.ToString() ?? "N/A"}'");

Console.ResetColor();

try

{

string newName = $"ChangedItem-{validDemoId + 1}";

Console.WriteLine($" Attempting to set Name to '{newName}'...");

itemToModify.GetType().GetProperty("Name")?.SetValue(itemToModify, newName);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($" Property 'Name' potentially updated (check via Show Container).");

IName? changedItem = container[validDemoId];

int changedIndex = changedItem != null ? FindIndexByReference(changedItem) : -1;

string changedIndexStr = changedIndex != -1 ? $"(Current Index: {changedIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Current value {changedIndexStr}: {changedItem?.ToString() ?? "Not Found!"}");

Console.ResetColor();

}

catch (TargetInvocationException tie) { PrintErrorMessage($" Error modifying property: {tie.InnerException?.Message ?? tie.Message}"); }

catch (Exception ex) { PrintErrorMessage($" Error modifying property: {ex.Message}"); }

}

else { Console.WriteLine($" Could not retrieve item with insertion ID {validDemoId + 1} for modification demonstration."); }

}

else

{

Console.WriteLine("\n3. Cannot demonstrate modification: No suitable item found with a valid insertion ID.");

}

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

// LinkedList Container Indexer Demonstration

static void DemonstrateIndexersList(ContainerLinkedList<IName> container, Random random)

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

if (container.Count == 0) return;

List<int> currentInsertionOrder = container.GetInsertionOrder();

if (currentInsertionOrder.Count == 0)

{

Console.WriteLine("Container has items but insertion order list is empty (unexpected). Cannot demonstrate.");

return;

}

// 1. Demonstrate Insertion ID Indexer (Get)

int randomIndexList = random.Next(currentInsertionOrder.Count);

int demoInsertionIdList = currentInsertionOrder[randomIndexList];

Console.WriteLine($"1. Accessing item by existing insertion ID [{demoInsertionIdList + 1}]:");

try

{

IName? itemById = container[demoInsertionIdList];

if (itemById != null)

{

int currentIndex = FindIndexByReference(itemById);

string indexStr = currentIndex != -1 ? $"(Current Index: {currentIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" Item with insertion ID {demoInsertionIdList + 1} not found (unexpected).");

Console.ResetColor();

}

}

catch (Exception ex)

{

PrintErrorMessage($" Error getting item by insertion ID {demoInsertionIdList + 1}: {ex.Message}");

}

// 2. Demonstrate Name Indexer (Get)

string? demoName = FindDemoName(container, random);

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

if (!string.IsNullOrWhiteSpace(demoName))

{

try

{

List<IName>? itemsByName = container[demoName];

if (itemsByName != null && itemsByName.Count > 0)

{

Console.ForegroundColor = ConsoleColor.Cyan;

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

foreach (var item in itemsByName)

{

int currentIndex = FindIndexByReference(item);

string indexStr = currentIndex != -1 ? $"(Index: {currentIndex + 1})" : "";

Console.WriteLine($" - {indexStr} {item.ToString() ?? "N/A"}");

}

Console.ResetColor();

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine($" No items found for name '{demoName}'.");

Console.ResetColor();

}

}

catch (Exception ex)

{

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

}

}

else

{

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

// 3. Demonstrate Insertion ID Indexer (Set)

int validDemoIdList = FindValidInsertionId(container);

if (validDemoIdList != -1)

{

Console.WriteLine($"\n3. Attempting to change item with insertion ID [{validDemoIdList + 1}] using property modification:");

IName? itemToModify = container[validDemoIdList];

if (itemToModify != null)

{

int currentIndex = FindIndexByReference(itemToModify);

string indexStr = currentIndex != -1 ? $"(Current Index: {currentIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Magenta;

Console.WriteLine($" Original Item {indexStr}: '{itemToModify.ToString() ?? "N/A"}'");

Console.ResetColor();

try

{

string newName = $"ChangedItem-{validDemoIdList + 1}";

Console.WriteLine($" Attempting to set Name to '{newName}'...");

itemToModify.GetType().GetProperty("Name")?.SetValue(itemToModify, newName);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($" Property 'Name' potentially updated (check via Show Container).");

IName? changedItem = container[validDemoIdList];

int changedIndex = changedItem != null ? FindIndexByReference(changedItem) : -1;

string changedIndexStr = changedIndex != -1 ? $"(Current Index: {changedIndex + 1})" : "";

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine($" Current value {changedIndexStr}: {changedItem?.ToString() ?? "Not Found!"}");

Console.ResetColor();

}

catch (TargetInvocationException tie) { PrintErrorMessage($" Error modifying property: {tie.InnerException?.Message ?? tie.Message}"); }

catch (Exception ex) { PrintErrorMessage($" Error modifying property: {ex.Message}"); }

}

else { Console.WriteLine($" Could not retrieve item with insertion ID {validDemoIdList + 1} for modification demonstration."); }

}

else

{

Console.WriteLine("\n3. Cannot demonstrate modification: No suitable item found with a valid insertion ID.");

}

Console.ForegroundColor = ConsoleColor.Yellow;

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

Console.ResetColor();

}

static string? FindDemoName(IName?[] items, int count, Random random)

{

string? demoName = null;

if (count > 0)

{

for (int i = 0; i < 5; ++i)

{

int randomIndex = random.Next(count);

IName? sourceItemForName = items[randomIndex];

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

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

}

}

return demoName;

}

static string? FindDemoName(ContainerLinkedList<IName> listContainer, Random random)

{

string? demoName = null;

if (listContainer.Count > 0)

{

for (int i = 0; i < 5; ++i)

{

int randomIndex = random.Next(listContainer.Count);

var node = listContainer.First;

int currentIndex = 0;

while (node != null && currentIndex < randomIndex)

{

node = node.Next;

currentIndex++;

}

if (node != null)

{

demoName = GetPropertyValue<string>(node.Data, "Name");

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

}

}

}

return demoName;

}

static int FindValidInsertionId(Container<IName> container)

{

int nextId = container.GetInsertionId();

if (nextId <= 0) return -1;

for (int id = nextId - 1; id >= 0; id--)

{

try

{

if (container[id] != null) return id;

}

catch (IndexOutOfRangeException) { /\* Ignore \*/ }

catch (Exception ex) { System.Diagnostics.Debug.WriteLine($"Debug: Error checking ID {id} in FindValidInsertionId (Array): {ex.Message}"); }

}

return -1;

}

static int FindValidInsertionId(ContainerLinkedList<IName> container)

{

List<int> order = container.GetInsertionOrder();

if (container.Count == 0 || order.Count == 0) return -1;

// Return the last added insertion ID (0-based)

return order[order.Count - 1];

}

// --- Property Modification Logic ---

static void ModifyProperty(object itemToModify, int itemInsertionId)

{

ArgumentNullException.ThrowIfNull(itemToModify);

var properties = itemToModify.GetType()

.GetProperties(BindingFlags.Public | BindingFlags.Instance)

.Where(p => p.CanWrite && p.GetSetMethod(true) != null)

.ToList();

if (properties.Count == 0)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("This object has no publicly 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 \*/ }

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;

if (!TryParseValue(newValueString, targetType, isNullable, out convertedValue))

{

return;

}

try

{

selectedProperty.SetValue(itemToModify, convertedValue, null);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"\nProperty '{selectedProperty.Name}' updated successfully for item (Insertion ID: {itemInsertionId + 1}).");

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

Console.ResetColor();

int currentIndex = FindIndexByReference((IName)itemToModify);

if (currentIndex != -1)

{

DisplayItemTable(currentIndex + 1, (IName)itemToModify);

}

else

{

PrintErrorMessage("Could not determine current index after modification for display.");

}

}

catch (TargetInvocationException tie)

{

PrintErrorMessage($"Validation Error setting property '{selectedProperty.Name}': {tie.InnerException?.Message ?? tie.Message}");

}

catch (ArgumentException argEx)

{

PrintErrorMessage($"Error setting property '{selectedProperty.Name}': Type mismatch or invalid argument. {argEx.Message}");

}

catch (Exception ex)

{

PrintErrorMessage($"Unexpected error setting property '{selectedProperty.Name}': {ex.Message}");

}

}

else

{

PrintErrorMessage("Invalid property choice.");

}

}

static bool TryParseValue(string input, Type targetType, bool isNullable, out object? parsedValue)

{

parsedValue = null;

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

{

return true;

}

try

{

if (targetType == typeof(bool))

{

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

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

parsedValue = true;

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

parsedValue = false;

else

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

}

else

{

TypeConverter converter = TypeDescriptor.GetConverter(targetType);

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

{

parsedValue = converter.ConvertFromString(null, CultureInfo.InvariantCulture, input);

}

else

{

parsedValue = Convert.ChangeType(input, targetType, CultureInfo.InvariantCulture);

}

}

return true;

}

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

{

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

return false;

}

}

// --- Automatic Generation ---

static void AutomaticGenerationArray(Container<IName> container, Random random, int count)

{

Console.WriteLine("Generating elements for Array Container...");

GenerateItems(count, random, item => container.Add(item));

Console.WriteLine("\nArray Generation process finished.");

}

static void AutomaticGenerationList(ContainerLinkedList<IName> container, Random random, int count)

{

Console.WriteLine("Generating elements for LinkedList Container...");

GenerateItems(count, random, item => container.AddLast(item));

Console.WriteLine("\nLinkedList Generation process finished.");

}

static void GenerateItems(int count, Random random, Action<IName> addAction)

{

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

{

IName newItem;

int typeChoice = random.Next(1, 9);

try

{

switch (typeChoice)

{

case 1: newItem = GenerateRandomProduct(random); break;

case 2: newItem = GenerateRandomRealEstate(random); break;

case 3: newItem = GenerateRandomRealEstateInvestment(random); break;

case 4: newItem = GenerateRandomApartment(random); break;

case 5: newItem = GenerateRandomHouse(random); break;

case 6: newItem = GenerateRandomHotel(random); break;

case 7: newItem = GenerateRandomLandPlot(random); break;

case 8: newItem = new RealEstate($"BaseRE{i}", random.Next(5000, 20000), $"Loc{i}", random.Next(50, 200), "Base"); break;

default: continue;

}

addAction(newItem);

Console.Write(".");

}

catch (Exception ex)

{

Console.Write("X");

System.Diagnostics.Debug.WriteLine($"\nGeneration Error: Failed to create item of type {typeChoice}. {ex.GetType().Name}: {ex.Message}");

}

}

}

// --- Manual Input ---

static void ManualInputArray(Container<IName> container)

{

IName? newItem = CreateItemManually();

if (newItem != null)

{

container.Add(newItem);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"{newItem.GetType().Name} added successfully to Array Container (Insertion ID: {container.GetInsertionId()}).");

Console.ResetColor();

}

}

static void ManualInputList(ContainerLinkedList<IName> container)

{

IName? newItem = CreateItemManually();

if (newItem != null)

{

container.AddLast(newItem);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"{newItem.GetType().Name} added successfully to LinkedList Container (Insertion ID: {container.GetNextInsertionId()}).");

Console.ResetColor();

}

}

static IName? CreateItemManually()

{

Console.ForegroundColor = ConsoleColor.Cyan;

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

Console.ForegroundColor = ConsoleColor.DarkCyan;

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 choice: ");

Console.ResetColor();

string classChoice = Console.ReadLine() ?? "";

try

{

return classChoice switch

{

"1" => CreateManualProduct(),

"2" => CreateManualRealEstate(),

"3" => CreateManualRealEstateInvestment(),

"4" => CreateManualApartment(),

"5" => CreateManualHouse(),

"6" => CreateManualHotel(),

"7" => CreateManualLandPlot(),

\_ => throw new ArgumentException("Invalid class choice.")

};

}

catch (ValueLessThanZero ex) { PrintErrorMessage($"Creation Error: {ex.Message}"); return null; }

catch (FormatException ex) { PrintErrorMessage($"Invalid input format during creation: {ex.Message}"); return null; }

catch (ArgumentException ex) { PrintErrorMessage($"Invalid argument during creation: {ex.Message}"); return null; }

catch (Exception ex) { PrintErrorMessage($"Unexpected error during creation: {ex.Message}"); return null; }

}

// --- Container Display ---

static void ShowContainerArray(Container<IName> container, int currentCount)

{

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

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Magenta;

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

Console.ResetColor();

PrintTableHeader(tableWidth);

IName?[] items = container.GetItems();

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

{

IName? item = items[i];

if (item == null) continue;

WriteDataRowByDisplayId(i + 1, item, tableWidth);

DrawHorizontalLine(tableWidth);

}

}

static void ShowContainerList(ContainerLinkedList<IName> container, int currentCount)

{

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

int tableWidth = CalculateTableWidth();

Console.ForegroundColor = ConsoleColor.Magenta;

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

Console.ResetColor();

PrintTableHeader(tableWidth);

var node = container.First;

int i = 0;

while (node != null)

{

IName item = node.Data;

if (item != null)

{

WriteDataRowByDisplayId(i + 1, item, tableWidth);

DrawHorizontalLine(tableWidth);

}

node = node.Next;

i++;

}

}

// Displays item using 1-based current index

static void DisplayItemTable(int displayId, IName item)

{

if (item == null) return;

int tableWidth = CalculateTableWidth();

PrintTableHeader(tableWidth);

WriteDataRowByDisplayId(displayId, item, tableWidth);

DrawHorizontalLine(tableWidth);

}

// --- Table Formatting Helpers ---

const int idWidth = 6;

const int classWidth = 16;

const int nameWidth = 18;

const int priceWidth = 16;

const int locationWidth = 20;

const int sizeWidth = 10;

const int typeWidth = 14;

const int marketValueWidth = 18;

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;

const int padding = 1;

const int numColumns = 17;

static int CalculateTableWidth()

{

int totalDataWidth = idWidth + classWidth + nameWidth + priceWidth + locationWidth + sizeWidth + typeWidth + marketValueWidth + investmentTypeWidth + floorWidth + hoaWidth + gardenWidth + poolWidth + roomsWidth + starWidth + soilWidth + infraWidth;

int totalPaddingWidth = numColumns \* padding;

return totalDataWidth + totalPaddingWidth;

}

static void PrintTableHeader(int tableWidth)

{

DrawHorizontalLine(tableWidth);

WriteHeaderRow();

DrawHorizontalLine(tableWidth);

}

static void WriteHeaderRow()

{

Console.ForegroundColor = ConsoleColor.Cyan;

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

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

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

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

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

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

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

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

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

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

Console.Write($"|{PadAndCenter("HOA", hoaWidth)}");

Console.Write($"|{PadAndCenter("Garden", gardenWidth)}");

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

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

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

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

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

Console.WriteLine("|");

Console.ResetColor();

}

static void WriteDataRowByDisplayId(int displayId, object item, int tableWidth)

{

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() ?? "-";

string FormatString(string? s) => string.IsNullOrWhiteSpace(s) ? "-" : s;

Type itemType = item.GetType();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Console.Write($"|{PadAndCenter(displayId.ToString(), idWidth)}");

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

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

Console.Write($"|{PadAndCenter(fPrice, priceWidth)}");

Console.Write($"|{PadAndCenter(loc, locationWidth)}");

Console.Write($"|{PadAndCenter(fSize, sizeWidth)}");

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

Console.Write($"|{PadAndCenter(fMktVal, marketValueWidth)}");

Console.Write($"|{PadAndCenter(invType, investmentTypeWidth)}");

Console.Write($"|{PadAndCenter(fFloor, floorWidth)}");

Console.Write($"|{PadAndCenter(fHoa, hoaWidth)}");

Console.Write($"|{PadAndCenter(fGarden, gardenWidth)}");

Console.Write($"|{PadAndCenter(fPool, poolWidth)}");

Console.Write($"|{PadAndCenter(fRooms, roomsWidth)}");

Console.Write($"|{PadAndCenter(fStar, starWidth)}");

Console.Write($"|{PadAndCenter(soil, soilWidth)}");

Console.Write($"|{PadAndCenter(fInfra, infraWidth)}");

Console.WriteLine("|");

}

static void DrawHorizontalLine(int tableWidth)

{

Console.ForegroundColor = ConsoleColor.DarkGray;

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

Console.ResetColor();

}

static string PadAndCenter(string? value, int totalWidth)

{

string val = value ?? "";

if (totalWidth <= 0) return "";

val = Truncate(val, totalWidth);

int spaces = totalWidth - val.Length;

int padLeft = spaces / 2;

return val.PadLeft(padLeft + val.Length).PadRight(totalWidth);

}

static string CenterString(string s, int width)

{

if (string.IsNullOrEmpty(s) || width <= 0) return new string(' ', Math.Max(0, width));

s = Truncate(s, width); // Ensure fits

int padding = Math.Max(0, (width - s.Length) / 2);

return new string(' ', padding) + s + new string(' ', Math.Max(0, width - s.Length - padding));

}

static string Truncate(string? value, int maxLength)

{

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

if (maxLength <= 0) return "";

if (value.Length <= maxLength) return value;

int subLength = Math.Max(0, maxLength - 3);

if (subLength == 0) return "...".Substring(0, Math.Min(3, maxLength));

return value.Substring(0, subLength) + "...";

}

// --- Reflection Property Getter ---

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

{

if (item == null) return default;

PropertyInfo? property = item.GetType().GetProperty(propertyName, BindingFlags.Public | BindingFlags.Instance);

if (property != null && property.CanRead)

{

try

{

object? value = property.GetValue(item);

if (value == null) return default;

if (value is TValue correctlyTyped) return correctlyTyped;

Type? underlyingTValue = Nullable.GetUnderlyingType(typeof(TValue));

if (underlyingTValue != null && underlyingTValue == property.PropertyType)

{

try { return (TValue)Convert.ChangeType(value, underlyingTValue, CultureInfo.InvariantCulture); } catch { /\* Ignore \*/ }

}

if (typeof(TValue) == typeof(string))

{

try { return (TValue)(object)Convert.ToString(value, CultureInfo.InvariantCulture)!; } catch { /\* Ignore \*/ }

}

else if (typeof(TValue) == typeof(decimal) && IsNumericType(property.PropertyType))

{

try { return (TValue)(object)Convert.ToDecimal(value, CultureInfo.InvariantCulture); } catch { /\* Ignore \*/ }

}

else if (typeof(TValue) == typeof(double) && IsNumericType(property.PropertyType))

{

try { return (TValue)(object)Convert.ToDouble(value, CultureInfo.InvariantCulture); } catch { /\* Ignore \*/ }

}

else if (typeof(TValue) == typeof(int) && IsNumericType(property.PropertyType))

{

try { return (TValue)(object)Convert.ToInt32(value, CultureInfo.InvariantCulture); } catch { /\* Ignore \*/ }

}

else if (typeof(TValue) == typeof(bool))

{

if (IsNumericType(property.PropertyType))

{

try { return (TValue)(object)(Convert.ToDouble(value, CultureInfo.InvariantCulture) != 0); } catch { /\* Ignore \*/ }

}

else if (property.PropertyType == typeof(string))

{

if (bool.TryParse((string)value, out bool boolVal)) return (TValue)(object)boolVal;

}

}

try { return (TValue)Convert.ChangeType(value, typeof(TValue), CultureInfo.InvariantCulture); } catch { /\* Ignore \*/ }

}

catch (Exception ex)

{

System.Diagnostics.Debug.WriteLine($"Reflection Error getting '{propertyName}': {ex.Message}");

}

}

return default;

}

private static bool IsNumericType(Type type)

{

if (type == null) return false;

switch (Type.GetTypeCode(type))

{

case TypeCode.Byte:

case TypeCode.Decimal:

case TypeCode.Double:

case TypeCode.Int16:

case TypeCode.Int32:

case TypeCode.Int64:

case TypeCode.SByte:

case TypeCode.Single:

case TypeCode.UInt16:

case TypeCode.UInt32:

case TypeCode.UInt64:

return true;

default:

return false;

}

}

// --- Container State Helpers ---

static bool IsContainerEmpty(out int count)

{

count = 0;

bool isEmpty = true;

if (activeContainerType == ContainerType.Array && containerArray != null)

{

isEmpty = containerArray.IsEmpty(false);

count = containerArray.GetCount();

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

count = containerList.Count;

isEmpty = (count == 0);

}

else

{

isEmpty = true;

}

if (isEmpty && activeContainerType != ContainerType.None)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("The active container is empty.");

Console.ResetColor();

}

else if (activeContainerType == ContainerType.None)

{

PrintErrorMessage("No container selected. Please use option 1 or 2 first.");

isEmpty = true;

}

return isEmpty;

}

static int GetActiveContainerCount()

{

if (activeContainerType == ContainerType.Array && containerArray != null)

{

return containerArray.GetCount();

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

return containerList.Count;

}

return 0;

}

// Gets the next insertion ID (0-based)

static int GetNextInsertionId()

{

if (activeContainerType == ContainerType.Array && containerArray != null)

{

return containerArray.GetInsertionId();

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

return containerList.GetNextInsertionId();

}

return 0;

}

// Finds the current 0-based index of an item

private static int FindIndexByReference(IName itemToFind)

{

if (itemToFind == null) return -1;

if (activeContainerType == ContainerType.Array && containerArray != null)

{

IName?[] currentItems = containerArray.GetItems();

int currentCount = containerArray.GetCount();

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

{

if (object.ReferenceEquals(currentItems[i], itemToFind))

{

return i;

}

}

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

var node = containerList.First;

int index = 0;

while (node != null)

{

if (object.ReferenceEquals(node.Data, itemToFind))

{

return index;

}

node = node.Next;

index++;

}

}

return -1;

}

private static int GetInsertionIdForItem(IName itemToFind)

{

if (itemToFind == null) return -1;

int index = FindIndexByReference(itemToFind);

if (index == -1) return -1;

try

{

if (activeContainerType == ContainerType.Array && containerArray != null)

{

int[] order = containerArray.GetInsertionOrder();

if (index < order.Length)

{

return order[index];

}

else { System.Diagnostics.Debug.WriteLine($"Warning: Index {index} out of bounds for insertion Order Array (Length: {order.Length})"); }

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

List<int> order = containerList.GetInsertionOrder();

if (index < order.Count)

{

return order[index];

}

else { System.Diagnostics.Debug.WriteLine($"Warning: Index {index} out of bounds for insertion Order List (Count: {order.Count})"); }

}

}

catch (Exception ex)

{

System.Diagnostics.Debug.WriteLine($"Error in GetInsertionIdForItem for index {index}: {ex.Message}");

}

return -1;

}

// Gets an item by Insertion ID from the active container

private static IName? GetItemByInsertionId(int insertionId)

{

try

{

if (activeContainerType == ContainerType.Array && containerArray != null)

{

return containerArray[insertionId];

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

return containerList[insertionId];

}

}

catch (IndexOutOfRangeException) { /\* ID not found or invalid for container \*/ }

catch (Exception ex) { PrintErrorMessage($"Unexpected error fetching item by insertion ID {insertionId + 1}: {ex.Message}"); }

return null;

}

private static IName? GetItemByCurrentIndex(int index)

{

if (index < 0) return null;

if (activeContainerType == ContainerType.Array && containerArray != null)

{

IName?[] items = containerArray.GetItems();

int count = containerArray.GetCount();

if (index < count)

{

return items[index];

}

}

else if (activeContainerType == ContainerType.LinkedList && containerList != null)

{

if (index < containerList.Count)

{

var node = containerList.First;

int i = 0;

while (node != null && i < index)

{

node = node.Next;

i++;

}

return node?.Data;

}

}

return null;

}

// --- 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.Max(0.01m, 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.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0, 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[] invTypes = { "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.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0m, Math.Round(marketValue, 2)), invTypes[random.Next(invTypes.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 floor = random.Next(1, 30);

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

return new Apartment(names[random.Next(names.Length)], Math.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0, Math.Round(size, 1)), types[random.Next(types.Length)], floor, Math.Max(0m, Math.Round(hoa, 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(-50, 1000) + random.NextDouble() \* 100;

bool pool = random.Next(3) == 0;

return new House(names[random.Next(names.Length)], Math.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0, Math.Round(size, 1)), types[random.Next(types.Length)], Math.Max(0.0, 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[] invTypes = { "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(20, 500);

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

return new Hotel(names[random.Next(names.Length)], Math.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0m, Math.Round(marketValue, 2)), invTypes[random.Next(invTypes.Length)], Math.Max(1, rooms), rating);

}

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[] invTypes = { "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 infra = random.Next(2) == 0;

return new LandPlot(names[random.Next(names.Length)], Math.Max(0.01m, Math.Round(price, 2)), locations[random.Next(locations.Length)], Math.Max(1.0m, Math.Round(marketValue, 2)), invTypes[random.Next(invTypes.Length)], soilTypes[random.Next(soilTypes.Length)], infra);

}

// --- Manual Creation Methods ---

static Product CreateManualProduct()

{

string name = ReadString("Enter Product Name: ");

decimal price = ReadDecimal("Enter Product Price (> 0): ", minValue: 0.01m);

return new Product(name, price);

}

static RealEstate CreateManualRealEstate()

{

string name = ReadString("Enter RealEstate Name: ");

decimal price = ReadDecimal("Enter RealEstate Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

double size = ReadDouble("Enter Size (> 0): ", minValue: 0.01);

string type = ReadString("Enter Type (e.g., Residential): ");

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

}

static RealEstateInvestment CreateManualRealEstateInvestment()

{

string name = ReadString("Enter Investment Name: ");

decimal price = ReadDecimal("Enter Investment Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

decimal marketValue = ReadDecimal("Enter Market Value (> 0): ", minValue: 0.01m);

string investmentType = ReadString("Enter Investment Type (e.g., REIT): ");

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

}

static Apartment CreateManualApartment()

{

string name = ReadString("Enter Apartment Name: ");

decimal price = ReadDecimal("Enter Apartment Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

double size = ReadDouble("Enter Size (> 0): ", minValue: 0.01);

string type = ReadString("Enter Type (e.g., Condo): ");

int floorNumber = ReadInt("Enter Floor Number (> 0): ", minValue: 1);

decimal hoaFees = ReadDecimal("Enter HOA Fees (>= 0): ", minValue: 0m);

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

}

static House CreateManualHouse()

{

string name = ReadString("Enter House Name: ");

decimal price = ReadDecimal("Enter House Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

double size = ReadDouble("Enter Size (> 0): ", minValue: 0.01);

string type = ReadString("Enter Type (e.g., Single-family): ");

double gardenSize = ReadDouble("Enter Garden Size (>= 0): ", minValue: 0.0);

bool pool = ReadBool("Has Pool (true/false/yes/no/1/0): ");

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

}

static Hotel CreateManualHotel()

{

string name = ReadString("Enter Hotel Name: ");

decimal price = ReadDecimal("Enter Hotel Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

decimal marketValue = ReadDecimal("Enter Market Value (> 0): ", minValue: 0.01m);

string investmentType = ReadString("Enter Investment Type: ");

int rooms = ReadInt("Enter Number of Rooms (> 0): ", minValue: 1);

int starRating = ReadInt("Enter Star Rating (1-5): ", minValue: 1, maxValue: 5);

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

}

static LandPlot CreateManualLandPlot()

{

string name = ReadString("Enter LandPlot Name: ");

decimal price = ReadDecimal("Enter LandPlot Price (> 0): ", minValue: 0.01m);

string location = ReadString("Enter Location: ");

decimal marketValue = ReadDecimal("Enter Market Value (> 0): ", minValue: 0.01m);

string investmentType = ReadString("Enter Investment Type: ");

string soilType = ReadString("Enter Soil Type (e.g., Loam): ");

bool infrastructureAccess = ReadBool("Has Infrastructure Access (true/false/yes/no/1/0): ");

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

}

// --- Robust Input Reading Helpers ---

static string ReadString(string prompt)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write(prompt);

Console.ResetColor();

return Console.ReadLine() ?? "";

}

static decimal ReadDecimal(string prompt, decimal? minValue = null, decimal? maxValue = null)

{

decimal value;

while (true)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write(prompt);

Console.ResetColor();

string? input = Console.ReadLine();

if (decimal.TryParse(input, NumberStyles.Any, CultureInfo.InvariantCulture, out value))

{

if ((minValue == null || value >= minValue) && (maxValue == null || value <= maxValue))

{

return value;

}

else

{

string minStr = minValue?.ToString("N2", CultureInfo.InvariantCulture) ?? "-infinity";

string maxStr = maxValue?.ToString("N2", CultureInfo.InvariantCulture) ?? "+infinity";

PrintErrorMessage($"Value must be{(minValue != null ? $" >= {minStr}" : "")}{(minValue != null && maxValue != null ? " and" : "")}{(maxValue != null ? $" <= {maxStr}" : "")}.");

}

}

else

{

PrintErrorMessage($"Invalid decimal format. Please use '.' as the decimal separator (e.g., 123.45). Input was: '{input}'");

}

}

}

static double ReadDouble(string prompt, double? minValue = null, double? maxValue = null)

{

double value;

while (true)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write(prompt);

Console.ResetColor();

string? input = Console.ReadLine();

if (double.TryParse(input, NumberStyles.Any, CultureInfo.InvariantCulture, out value))

{

if ((minValue == null || value >= minValue) && (maxValue == null || value <= maxValue))

{

return value;

}

else

{

string minStr = minValue?.ToString("N1", CultureInfo.InvariantCulture) ?? "-infinity";

string maxStr = maxValue?.ToString("N1", CultureInfo.InvariantCulture) ?? "+infinity";

PrintErrorMessage($"Value must be{(minValue != null ? $" >= {minStr}" : "")}{(minValue != null && maxValue != null ? " and" : "")}{(maxValue != null ? $" <= {maxStr}" : "")}.");

}

}

else

{

PrintErrorMessage($"Invalid number format. Please use '.' as the decimal separator (e.g., 12.3). Input was: '{input}'");

}

}

}

static int ReadInt(string prompt, int? minValue = null, int? maxValue = null)

{

int value;

while (true)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write(prompt);

Console.ResetColor();

string? input = Console.ReadLine();

if (int.TryParse(input, NumberStyles.Integer, CultureInfo.InvariantCulture, out value))

{

if ((minValue == null || value >= minValue) && (maxValue == null || value <= maxValue))

{

return value;

}

else

{

string minStr = minValue?.ToString(CultureInfo.InvariantCulture) ?? "any";

string maxStr = maxValue?.ToString(CultureInfo.InvariantCulture) ?? "any";

PrintErrorMessage($"Value must be between {minStr} and {maxStr}.");

}

}

else

{

PrintErrorMessage($"Invalid integer format. Input was: '{input}'");

}

}

}

static bool ReadBool(string prompt)

{

while (true)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write(prompt);

Console.ResetColor();

string input = Console.ReadLine()?.Trim().ToLowerInvariant() ?? "";

if (input == "true" || input == "1" || input == "yes" || input == "y") return true;

if (input == "false" || input == "0" || input == "no" || input == "n") return false;

PrintErrorMessage("Invalid boolean input. Use true/false/yes/no/1/0.");

}

}

}