«Создание библиотеки утилит для работы с массивами»

Предисловие: От монолитного кода к модульному проектированию

До этого момента наши программы, как правило, представляли собой единый поток инструкций в методе `Main`. Такой подход может работать для "простых" задач, но слабо применим в реальной разработке. Более серьезные проекты и программы строятся из небольших, независимых и повторно используемых компонентов - **функций (методов)**.

Эта лабораторная работа посвящена написанию таких компонентов. В рамках этой работы мы создадим цельный инструмент: небольшую статическую библиотеку утилит для работы с массивами. На её примере мы изучим ключевые аспекты работы с функциями: способы передачи и возврата данных, модификацию внешних переменных и передачу в качестве параметров не только данных, но и самой логики выполнения.

Цель: Научиться проектировать и использовать функции, понять разницу между способами передачи параметров (`value`, `ref`, `out`, `params`) и познакомиться с мощью делегатов для создания гибкого и расширяемого кода.

Задания:

Все создаваемые методы должны быть статическими (`static`) и находиться в одном классе ("рядом" с методом `Main`).

Задание 0. Начало работы с функциями

Задача 1:

Напишите две функции `Max2(int a, int b)` и `Min2(int a, int b)`, возвращающие максимум и минимум из двух целых чисел. Пара чисел вводится из консоли, результат работы обеих функций нужно вывести в консоль.

Демонстрация в `Main`:
Вызовите обе функции.
Пример:

```
Вход: 7, -3

Выход: Max = 7, Min = -3
```

Задание 1. Универсальный анализатор (`params` и перегрузка)

Задача 1:

Написать две **перегруженные** версии метода `GetStats`, который анализирует произвольный набор чисел.

1. `public static string GetStats(params int[] numbers)`
2. `public static string GetStats(params double[] numbers)`

Зачем?
Изучение способа создания функций, принимающих переменное количество аргументов с помощью ключевого слова `params`.
Изучить, как с помощью **перегрузки методов** один метод может иметь несколько реализаций для работы с разными типами данных (пример: `int` и `double`).

Реализация:
Каждый метод должен вычислять: количество элементов, их сумму, среднее арифметическое, а также максимальный и минимальный элементы.
Результат должен возвращаться (`return`) в виде одной, отформатированной для вывода строки.

Демонстрация в `Main`:
Вызовите обе версии метода с разным количеством аргументов.
Пример:

1.

```
GetStats(1, 2, 3)

Кол-во: 3;
```

```
Сумма: 6;
Ср. ариф: 2;
Макс: 3;
Мин: 1;
```

2.

```
GetStats(10, 20, 30, 40, 50)

Кол-во: 5;
Сумма: 150;
Ср. ариф: 30;
Макс: 50;
Мин: 10;
```

3.

```
GetStats(1.1, 2.2, -5.5)

Кол-во: 3;
Сумма: -2.2;
Ср. ариф: -0.73;
Макс: 2.2;
Мин: -5.5;
```

Задание 2. Передача аргументов по ссылке (`ref` для ссылочных типов)

Задача 1:

Написать метод `Swap`, который принимает значения по ссылке, меняет местами значения двух переменных без использования дополнительных структур данных.

```
public static void Swap(ref int a, ref int b)
```

Реализация:
Метод должен менять местами значения переменных, которые были переданы в качестве аргумента из `Main`.

Демонстрация в `Main`:
Создайте 2 переменные, выведите на экран эти переменные, вызовите метод `Swap`, а затем снова выведите **эти же** переменные в том же порядке, что и первый раз, для демонстрации, что переменные "поменялись местами".

Зачем:
Передача аргументов по ссылке: Освоить механизм передачи переменных по ссылке с помощью ключевого слова `ref`, понять отличие передачи по значению и по ссылке, а также научиться изменять значения переменных вне вызывающего метода.

Задача 2:

Написать метод `Reverse`, который инвертирует порядок элементов в массиве, изменяя его "на месте".

```
public static void Reverse(int[] array)
```

Реализация:
Метод не должен создавать новый массив. Он должен изменять тот массив, который был передан в качестве аргумента, меняя местами его элементы (первый с последним, второй с предпоследним и т.д.).

Демонстрация в `Main`:
Создайте массив, выведите его на экран, вызовите метод `Reverse`, а затем снова выведите **этот же** массив, чтобы продемонстрировать, что его содержимое изменилось.

Цели обучения:
Передача ссылочных типов: Понимание того, что при передаче массива в метод по умолчанию передаётся **копия ссылки**, но эта ссылка указывает на **тот же самый объект** в памяти. Следовательно, изменения *содержимого* объекта (элементов массива) видны вне метода.

Задача 3:

Написать метод `Reverse`, который возвращает **новый** массив с элементами в обратном порядке, не изменяя исходный массив.

```
public static int[] Reverse(int[] array)
```

Реализация:
Метод должен создавать новый массив, в который копируются элементы исходного массива в обратном порядке. Исходный массив не должен изменяться.

Демонстрация в `Main`:
Создайте массив, выведите его на экран, вызовите метод `Reverse`, а затем выведите **новый** массив, чтобы показать, что его содержимое - это инвертированная версия исходного массива, а сам исходный массив остался без изменений.

Цели обучения:
Работа с массивами и возврат новых объектов: Освоить создание новых объектов на основе существующих данных, а также понять разницу между изменением исходных данных и созданием новых структур.

Задание 3. Безопасный поиск с несколькими результатами (`out`)

Задача:

Написать метод `TryFind`, который выполняет безопасный поиск элемента в массиве.

```
public static bool TryFind(int[] array, int value, out int index)
```

Реализация:
Метод ищет значение `value` в массиве `array`.

- Если элемент найден, метод присваивает его позицию выходному параметру `index` и возвращает `true`.
- Если элемент не найден, метод присваивает `index` значение `-1` или `null` и возвращает `false`.

Демонстрация в `Main`:
Продемонстрируйте оба сценария (когда элемент найден и когда нет), используя результат работы метода в условной конструкции `if (TryFind(...)) { ... }`.

Зачем:

- `out`**-параметры:** Изучение идиоматичного для C# способа возврата из функции нескольких значений (в данном случае — логического

результата `bool` и индекса `int`).

Задание 4. Поведение как параметр (Делегаты)

Задача:

Написать метод для фильтрации массивов, который может применять любой заданный критерий.

1. **Объявите тип делегата:**
   `public delegate bool FilterCondition(int number);`
   Этот делегат определяет сигнатуру для всех методов, которые могут использоваться в качестве критерия фильтрации.
2. **Напишите универсальный фильтр:**
   `public static int[] Filter(int[] sourceArray, FilterCondition condition)`
   Этот метод должен итерировать по массиву `sourceArray` и возвращать **новый** массив, содержащий только те элементы, для которых вызов `condition(element)` вернул `true`.
3. **Создайте методы, соответствующие делегату:**

- `public static bool IsEven(int n) { /* ... */ }` // Проверяет на четность
- `public static bool IsPositive(int n) { /* ... */ }` // Проверяет на положительность
- *Создайте еще один собственный метод-критерий.*

4. **Демонстрация в `Main`:**
   Создайте исходный массив. Вызовите универсальный метод `Filter` несколько раз, передавая в него разные методы-критерии (`IsEven`, `IsPositive`), и выведите результаты фильтрации на экран.

Цели обучения:

- **Делегаты:** Понимание фундаментального принципа, согласно которому в метод можно передавать не только данные, но и **поведение (логику)** в виде другого метода. Это открывает путь к созданию гибкого, расширяемого и мощного кода.

Задание 5. Ref для массивов

Задача:

Реализовать методы добавления значения в конец массива. Программа должна выполнился без ошибок и её вывод показал разницу в работе двух подходов.

Описание

В данном задании вам нужно на практике объяснить фундаментальную разницу между передачей массива по значению (когда передается копия ссылки) и по ссылке (с помощью ref). Для этого вам нужно реализовать два "похожих" метода и проанализировать их поведение.

Ниже представлен готовый `Main`, который служит лабораторным стендом для нашего эксперимента. Он последовательно вызывает оба метода и демонстрирует результат их работы.

Обратите внимание на вывод `GetHashCode()`. Хэш-код переменной меняется, если переменная указывает на новый объект в памяти. Если остался прежним - объект тот же.

Пример кода:

```
internal class Program
{
    static void Main(string[] args)
    {
        int[] array = { 1, 2, 3 };

        Console.WriteLine("--- ЭКСПЕРИМЕНТ 1: Попытка изменить
ссылку БЕЗ ref ---");
        Console.WriteLine($"Массив ДО вызова: [{string.Join(", ",
array)}]. Хэш-код: {array.GetHashCode()}");

        // Вызываем метод, который должен был бы подменить массив,
но не сможет
        BrokenSafeAppend(array, 4);

        Console.WriteLine($"Массив ПОСЛЕ вызова: [{string.Join(",
", array)}]. Хэш-код: {array.GetHashCode()}");
        Console.WriteLine("ОЖИДАНИЕ: Массив должен был измениться.
РЕАЛЬНОСТЬ: Он не изменился.\n");
```

```
        Console.WriteLine("--- ЭКСПЕРИМЕНТ 2: Изменение ссылки с помощью ref ---");
        Console.WriteLine($"Массив ДО вызова: [{string.Join(", ", array)}]. Хэш-код: {array.GetHashCode()}");

        // А вот этот вызов сработает как надо
        RefSafeAppend(ref array, 4);

        Console.WriteLine($"Массив ПОСЛЕ вызова: [{string.Join(", ", array)}]. Хэш-код: {array.GetHashCode()}");
        Console.WriteLine("ОЖИДАНИЕ: Массив изменился. РЕАЛЬНОСТЬ: Успех!\n");

        // КОНТРОЛЬНЫЕ ВОПРОСЫ:
        // 1. Почему результаты двух экспериментов отличаются?
        // 2. Что конкретно делает ключевое слово 'ref' в сигнатуре метода 'RefSafeAppend'?
        // 3. Почему хэш-код массива изменился во втором эксперименте, но не в первом? Что это доказывает?
    }

    // ЗАДАНИЕ 1: Реализуйте этот метод. Он должен создавать новый массив, копировать в него старые элементы и добавлять новый.
    // Ключевое слово ref здесь ГАРАНТИРУЕТ, что присваивание 'sourceArray = newArray' повлияет на переменную 'array' в Main.
    public static void RefSafeAppend(ref int[] sourceArray, int value)
    {
        // ВАШ КОД ЗДЕСЬ
        // 1. Создайте newArray нужного размера
        // 2. Скопируйте данные
        // 3. Добавьте value
        // 4. Сделайте так, чтобы sourceArray теперь указывал на newArray
    }

    // ЗАДАНИЕ 2: Скопируйте сюда РАБОЧУЮ реализацию из RefSafeAppend, но УБЕРИТЕ ключевое слово ref из сигнатуры.
    // Этот метод теперь будет работать с КОПИЕЙ ССЫЛКИ и не сможет изменить оригинальный массив в Main.
    public static void BrokenSafeAppend(int[] sourceArray, int value)
    {
```

```
        // ВАШ КОД ЗДЕСЬ
        // Он будет идентичен коду выше, но работать будет только
внутри этого метода.
    }

}
```

Контрольные вопросы:

Во время защиты вам нужно дать ответы на следующие вопросы:

1. Почему результаты двух экспериментов в `Main` отличаются, хотя внутренняя логика методов идентична?
2. Что конкретно делает ключевое слово `ref` в сигнатуре метода `RefSafeAppend`, чего не происходит в `BrokenSafeAppend`?
3. Почему хэш-код массива изменился во втором эксперименте, но не в первом? Что это показывает?