«Динамический массив: реализация собственного аналога `List<T>`»

А зачем мы создаем то, что уже существует?

В стандартной библиотеке .NET есть класс `List<T>`. Это готовый оптимизированный контейнер, который позволяет добавлять элементы, не *задумываясь* о его размере. Он кажется *"бесконечным"*. Однако на аппаратном уровне не существует бесконечных массивов.

`List<T>` **— это высокоуровневая абстракция.** Под его удобным интерфейсом скрывается обычный статический массив, который имеет фиксированный размер.

Цель этой лабораторной работы — посмотреть на "механизмы бесконечного списка" и реализовать их самостоятельно.

В этой работе мы рассмотрим внутреннее устройство списка, какие операции по выделению памяти и копированию данных стоят за простым вызовом метода `Add()`, а самое главное: мы перейдем от использования готового инструмента к созданию своего (да, на основе существующего, но учиться же надо).

Задание:

Создать консольное приложение и в нём реализовать класс `MyArrayList`, который будет упрощенным аналогом `List<int>`.

Часть 1. Структура класса

Создайте класс `MyArrayList`. Он должен содержать следующие компоненты:

1. **Приватные поля (внутреннее состояние):**
   - `private int[] _data;` - статический массив, который будет служить хранилищем для элементов.
   - `private int _count;` - целочисленный счётчик, отслеживающий количество реально добавленных элементов.

- `private int defaultCapacity` - стандартный размер массива при создании списка. **Выбрать самостоятельно.**

2. **Публичные свойства (внешний интерфейс):**
   - `public int Count => _count;` — свойство "только для чтения", возвращающее текущее количество элементов в списке.
   - `public int Capacity => _data.Length;` — свойство "только для чтения", показывающее текущую ёмкость (вместимость, размер) внутреннего массива.
3. **Конструктор (`public MyArrayList()`):**
   - При создании нового объекта `MyArrayList` конструктор должен инициализировать внутренний массив `_data` некоторым начальным размером (`defaultCapacity`) и установить счётчик `_count` в значение 0.

Часть 2. Основной функционал

Метод Add

Реализуйте главный публичный метод для добавления элементов: `public void Add(int item)`

Метод Remove

Реализуйте публичный метод для удаления элемента из коллекции (первого найденного): `public void Remove(int item)`

Метод RemoveAt

Реализуйте публичный метод для удаления элемента по индексу: `public void RemoveAt(int index)

Метод Insert

Реализуйте публичный метод для вставки элемента по указанному индексу: `public void Insert(int index, int item)`

Метод Clear

Реализуйте публичный метод для очистки списка: `public void Clear()`

Метод IndexOf

Реализуйте публичный метод для получения индекса первого найденного элемента: `public int IndexOf(int item)`

Часть 2.5. Вспомогательные приватные методы

Метод Grow

Реализуйте приватный метод, увеличивающий емкость списка, когда в нем заканчивается место: `private void Grow()`

Метод CheckIndexForAccess

Реализуйте приватный метод, проверяющий переданный индекс на корректность значений: `private void CheckIndexForAccess(int index)`

Часть 3. Доступ к элементам (Индексатор)

Для того чтобы сделать наш класс похожим на стандартные коллекции и массивы, реализуйте **индексатор**: `public int this[int index]`

Тестирование реализации (Метод `Main`)

В методе `Main` вашего приложения напишите код для проверки функциональности класса `MyArrayList`:

Базовый минимум

1. Создайте экземпляр `MyArrayList`.
2. В цикле добавьте 3-4 элемента. После каждого добавления выводите на экран значения свойств `Count` и `Capacity`, чтобы отслеживать состояние объекта.
3. Продолжайте добавлять элементы до тех пор, пока не будет вызван механизм `Resize`. Для наглядности добавьте в метод `Resize` вывод в консоль сообщения, сигнализирующего о его вызове.
4. Продемонстрируйте работу индексатора: выведите на экран несколько элементов, обращаясь к ним по корректным индексам.
5. Продемонстрируйте работу механизма защиты от неверного индекса. Поместите обращение к элементу по заведомо

некорректному индексу (например, `list[99]`) в блок `try...catch` и обработайте ожидаемое исключение `IndexOutOfRangeException`.

Проверка конструктора

1. Вызов `new MyArrayList(0)` должен создать список с ёмкостью 0. Первый `Add` должен вызвать `Grow`.
2. Вызов `new MyArrayList(-1)` должно кидать исключение типа `ArgumentOutOfRangeException` (Нельзя создать массив отрицательной длины).

Операции на пустом списке

1. Вызов функции `RemoveAt(0)` в пустом списке должен кидать исключение `IndexOutOfRangeException` (индекс должен быть валидным).
2. Вызов функции `Remove(123)` в пустом списке **не должен** кидать исключение.

Работа с индексами

1. Вставка в конец списка (`Insert(Count, item)`) должна вставлять элемент в конец списка.
2. Вставка в индекс после максимально возможного (`Insert(Count + 1, item)`) и вставка в отрицательный индекс (`Insert(-1, item)`) должны кидать исключение `ArgumentOutOfRangeException`.
3. Продемонстрировать удаление последнего элемента (`RemoveAt(Count - 1)`), вызов `RemoveAt(Count)` должно кидать исключение.
4. Вызов функции `IndexOf(-1)` должен возвращать значение `-1`, не исключение.