

# Обобщенные типы и итераторы

## 1. Реализация динамического массива

До этого момента мы работали с массивами ( `int[]` , `string[]` и т.д.). Главное ограничение массивов в `C#` состоит в том, что их **длина фиксирована**. Мы должны заранее знать, сколько элементов нужно сохранить.

Но часто мы не знаем заранее, сколько элементов должно быть в массиве. Поэтому нам нужен **динамический массив**, который изменяет свой размер, при добавлении и удалении элементов.

В `C#` для этого уже есть готовый класс `List<T>` , но мы пока сделаем свой собственный — чтобы понять, как он устроен внутри.

Сначала ограничимся простейшим случаем: список чисел ( `int` ). Мы создадим класс `MyListInt` , внутри которого будем хранить обычный массив, а при переполнении увеличивать его размер.

### Идея реализации

1. Внутри класса есть обычный массив `int[] items` .
2. Есть поле `count` , которое хранит текущее количество элементов.
3. Метод `Add` добавляет новый элемент в конец массива.
  - Если массив переполнен, создаём новый массив в 2 раза больше и копируем туда все элементы.
4. Метод `RemoveAt(int index)` удаляет элемент по индексу.
5. Индексатор `this[int index]` даёт доступ к элементу, как в обычном массиве.

## Пример

```
class MyListInt
{
    private int[] items;    // внутренний массив
    private int count;      // текущее количество элементов

    public MyListInt()
    {
        items = new int[4]; // начальный размер
        count = 0;
    }
    // Количество элементов в списке
    public int Count => count;

    // Добавление элемента
    public void Add(int value)
    {
        if (count == items.Length)
            Resize();

        items[count] = value;
        count++;
    }
    // Удаление по индексу
    public void RemoveAt(int index)
    {
        if (index < 0 || index >= count)
            throw new ArgumentOutOfRangeException();

        for (int i = index; i < count - 1; i++)
        {
```

```

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

    count--;
}
// Индексатор (доступ как к массиву)
public int this[int index]
{
    get
    {
        if (index < 0 || index >= count)
            throw new ArgumentOutOfRangeException();
        return items[index];
    }
    set
    {
        if (index < 0 || index >= count)
            throw new ArgumentOutOfRangeException();
        items[index] = value;
    }
}
// Увеличение массива в 2 раза
private void Resize()
{
    int newSize = items.Length * 2;
    int[] newArray = new int[newSize];
    for (int i = 0; i < items.Length; i++)
        newArray[i] = items[i];
    items = newArray;
}
}

```

## Пример работы

```
var list = new MyListInt();  
list.Add(10);  
list.Add(20);  
list.Add(30);  
list.Add(40);  
list.Add(50); // здесь массив увеличится автоматически  
  
Console.WriteLine(list.Count); // 5  
Console.WriteLine(list[2]);    // 30  
  
list.RemoveAt(1);              // удалим число 20  
Console.WriteLine(list[1]);     // теперь это 30
```

# Универсальность и проблема типизации

Мы сделали свой `MyListInt` , который отлично работает с числами.

Но что будет, если нам нужен список строк? Или список чисел с плавающей точкой ( `double` )?

Тогда придётся писать отдельные классы:

```
class MyListString { /* всё то же самое, но со string */ }  
class MyListDouble { /* то же самое, но с double */ }
```

Очевидно, это неудобно:

- код приходится дублировать;
- любые исправления придётся вносить во все версии;
- класс становится негибким.

## Попытка решения через `object`

В C# все типы (и значения, и ссылки) в конечном счёте наследуются от `object` .

Значит, можно было бы хранить внутри списка **массив** `object[]` , и тогда в один список можно добавлять всё подряд:

## Пример

```
class MyListObject {
    private object[] items;
    private int count;
    public MyListObject() {
        items = new object[4];
        count = 0;
    }
    public void Add(object value) {
        if (count == items.Length) Resize();
        items[count] = value;
        count++;
    }
    public object this[int index] => items[index];
    private void Resize() {
        object[] newArray = new object[items.Length * 2];
        for (int i = 0; i < items.Length; i++)
            newArray[i] = items[i];
        items = newArray;
    }
}
```

Теперь можно сделать так:

```
var list = new MyListObject();
list.Add(10);
list.Add("Hello");
list.Add(3.14);
```

Список стал универсальным. Но у этого решения есть **серьёзная проблема**.

# Проблема

Когда мы достаём элемент, он имеет тип `object`.

Чтобы использовать его по назначению, нужно сделать **приведение типа**:

```
int number = (int)list[0];    // работает
string text = (string)list[1]; // работает
double pi = (double)list[2];  // работает
```

Но что, если мы ошиблись?

```
int error = (int)list[1]; // InvalidCastException во время выполнения!
```

Компилятор не может проверить такие ошибки на этапе компиляции, потому что все элементы — это просто `object`.

Значит, баги проявятся только во время запуска программы.

Такое решение небезопасно.

## 2. Обобщенные типы (Generics)

Чтобы избежать дублирования кода и ошибок времени выполнения, в C# есть **обобщённые типы** (*generics*).

### Определение:

Обобщённые типы позволяют писать один класс или метод, который работает с разными типами данных, оставаясь при этом **типобезопасным**.

Другими словами, мы описываем алгоритм один раз, а компилятор сам подставляет конкретные типы.

### Пример: универсальный список

```
class MyList<T>
{
    private T[] items;
    private int count;

    public MyList()
    {
        items = new T[4];
        count = 0;
    }
    public int Count => count;
    public void Add(T value)
    {
        if (count == items.Length)
            Resize();
        items[count] = value;
        count++;
    }
    public T this[int index] => items[index];
}
```



```

private void Resize()
{
    T[] newArray = new T[items.Length * 2];
    for (int i = 0; i < items.Length; i++)
        newArray[i] = items[i];
    items = newArray;
}
}

```

Теперь мы можем использовать один и тот же класс для разных типов данных:

```

var intList = new MyList<int>();
intList.Add(10);
intList.Add(20);
Console.WriteLine(intList[0]); // 10

var stringList = new MyList<string>();
stringList.Add("Hello");
stringList.Add("World");
Console.WriteLine(stringList[1]); // World

var pointList = new MyList<Point>();
pointList.Add(new Point { X = 1, Y = 2 });

```

- `MyList<int>` — работает только с числами `int`.
- `MyList<string>` — только со строками.
- `MyList<Point>` — с нашими объектами `Point`.

При этом код списка **один и тот же**.

# Почему обобщенные типы лучше чем `object`

## 1. Безопасность типов

Ошибки приводятся на этапе компиляции:

```
var list = new MyList<int>();  
list.Add("text"); // ошибка компиляции
```

## 2. Нет затрат на приведение типов (casting)

При использовании `object` приходится постоянно делать downcast, а это замедляет программу.

Дженерики работают без таких накладных расходов.

## 3. Универсальность без дублирования

Код пишется один раз, а компилятор генерирует версии для каждого типа.

Таким образом, дженерики объединяют **удобство** (как у `object`) и **безопасность** (как у специализированных классов).

## Синтаксис и возможности обобщенных типов

Мы уже сделали универсальный класс `MyList<T>`, который работает с любыми типами. Но generics в C# дают намного больше возможностей, чем просто обобщённые коллекции.

## Обобщённые методы

Помимо классов, generic-параметры можно использовать и в методах.

Пример: метод, который меняет местами два значения:

```
static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Использование:

```
int x = 10, y = 20;
Swap(ref x, ref y);
Console.WriteLine($"{x}, {y}"); // 20, 10

string s1 = "Hello", s2 = "World";
Swap(ref s1, ref s2);
Console.WriteLine($"{s1}, {s2}"); // World, Hello
```

Обратите внимание: метод один, но работает с любыми типами (`int`, `string`, `Point` и т.д.). Компилятор сам подставляет тип в `<T>` во время вызова.

## Несколько параметров

У generics может быть не один, а несколько параметров.

Например, сделаем простейшую пару значений:

```
class Pair<TFirst, TSecond>
{
    public TFirst First { get; set; }
    public TSecond Second { get; set; }

    public Pair(TFirst first, TSecond second)
    {
        First = first;
        Second = second;
    }
}
```

Использование:

```
var pair1 = new Pair<int, string>(1, "One");
Console.WriteLine($"{pair1.First} - {pair1.Second}");

var pair2 = new Pair<string, bool>("Active", true);
Console.WriteLine($"{pair2.First} = {pair2.Second}");
```

Мы получаем удобную структуру для хранения двух связанных значений, но при этом они могут быть любого типа.

## Ограничения ( where )

Иногда нужно ограничить, какие типы можно использовать.

Для этого есть ключевое слово `where`.

### Пример 1. Ограничение по ссылочному типу

```
class MyRepository<T> where T : class
{
    public void Save(T item) { /* ... */ }
}
```

Теперь `T` должен быть ссылочным типом:

```
var repo1 = new MyRepository<string>(); // ок
var repo2 = new MyRepository<int>();    // ошибка: int — значимый тип
```

---

## Пример 2. Ограничение по интерфейсу

```
class MyComparer<T> where T : IComparable<T>
{
    public bool IsGreater(T a, T b) => a.CompareTo(b) > 0;
}
```

Использование:

```
var cmp = new MyComparer<int>();
Console.WriteLine(cmp.IsGreater(10, 5)); // true
```

---

## Пример 3. Ограничение наличием конструктора

```
class Factory<T> where T : new()
{
    public T Create() => new T();
}
```

Теперь можно создавать объекты типа `T`, но только если у них есть конструктор без параметров.

## 3. IEnumerable и IEnumerator

### Перебор коллекции

Давайте вспомним, какие циклы есть в языке C# и в каких ситуациях каждый из них нужно использовать:

1. `for` — когда есть явный счётчик.
2. `foreach` — когда нам нужно просто перебрать все элементы в коллекции.
3. `while` — во всех остальных случаях.

Если мы попробуем использовать созданный нами динамический массив `MyList<T>` в конструкции `foreach`, то получим ошибку компиляции:

```
var list = new MyList<int>();  
list.Add(10);  
list.Add(20);  
  
foreach (var item in list) // Ошибка!  
{  
    Console.WriteLine(item);  
}
```

### Почему так?

Компилятор требует, чтобы тип поддерживал **перечисление элементов** через специальные интерфейсы.

## IEnumerable и IEnumerator

- **IEnumerator** — интерфейс, который говорит:

«Объект можно перечислить».

В нём есть метод `GetEnumerator()`, который возвращает объект-итератор.

- **IEnumerator** — интерфейс самого **итератора** (обходчика).

В нём есть:

- свойство `Current` — текущий элемент,
- метод `MoveNext()` — переход к следующему элементу (возвращает `true`, если ещё есть элементы),
- метод `Reset()` (обычно не используется).

Конструкция `foreach` «под капотом» работает так:

1. Вызывает `GetEnumerator()` у коллекции.
2. В цикле вызывает `MoveNext()`.
3. Берёт `Current`.



## Реализация в MyList<T>

Сделаем MyList<T> перечисляемым:

```
class MyList<T> : IEnumerable<T>
{
    private T[] _items;
    private int _count;

    public MyList()
    {
        _items = new T[4];
        _count = 0;
    }

    public void Add(T item)
    {
        if (_count == _items.Length)
        {
            Array.Resize(ref _items, _items.Length * 2);
        }
        _items[_count++] = item;
    }

    public int Count => _count;

    // Реализация IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        return new MyEnumerator(this);
    }
}
```

```
IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

// Внутренний класс-итератор
private class MyEnumerator : IEnumerator<T>
{
    private MyList<T> _list;
    private int _index;

    public MyEnumerator(MyList<T> list)
    {
        _list = list;
        _index = -1; // перед первым элементом
    }

    public T Current => _list._items[_index];

    object IEnumerator.Current => Current;

    public bool MoveNext()
    {
        _index++;
        return _index < _list._count;
    }

    public void Reset()
    {
        _index = -1;
    }

    public void Dispose() { }
}
}
```

## Использование

Теперь `MyList<T>` можно перебирать в `foreach`:

```
var numbers = new MyList<int>();  
numbers.Add(1);  
numbers.Add(2);  
numbers.Add(3);  
  
foreach (var n in numbers)  
{  
    Console.WriteLine(n);  
}
```

Вывод:

```
1  
2  
3
```

## 4. yield return

Как вы могли заметить, реализация `IEnumerable` и `IEnumerator` занимает много места: отдельный класс-итератор, поля, `MoveNext`, `Reset`, `Dispose` и т.д.

Для упрощения в C# придумали специальный механизм: `yield return`.

### Что делает `yield return`

- Позволяет описывать **последовательность элементов** простым способом.
- Компилятор сам «за кулисами» создаёт класс-итератор и реализует `IEnumerator`.
- Каждый `yield return` возвращает **один элемент коллекции**.
- Когда цикл `foreach` снова запрашивает элемент, метод **продолжается с того же места**, где он был остановлен.

### Реализация в `MyList<T>`

Раньше мы писали вручную класс `MyEnumerator`.

Теперь можно упростить `GetEnumerator()`:

```
class MyList<T> : IEnumerable<T>
{
    private T[] _items;
    private int _count;

    public MyList()
    {
        _items = new T[4];
        _count = 0;
    }
}
```

```

public void Add(T item)
{
    if (_count == _items.Length)
    {
        Array.Resize(ref _items, _items.Length * 2);
    }
    _items[_count++] = item;
}

public int Count => _count;

// Реализация IEnumerable<T> через yield return
public IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < _count; i++)
    {
        yield return _items[i];
    }
}

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

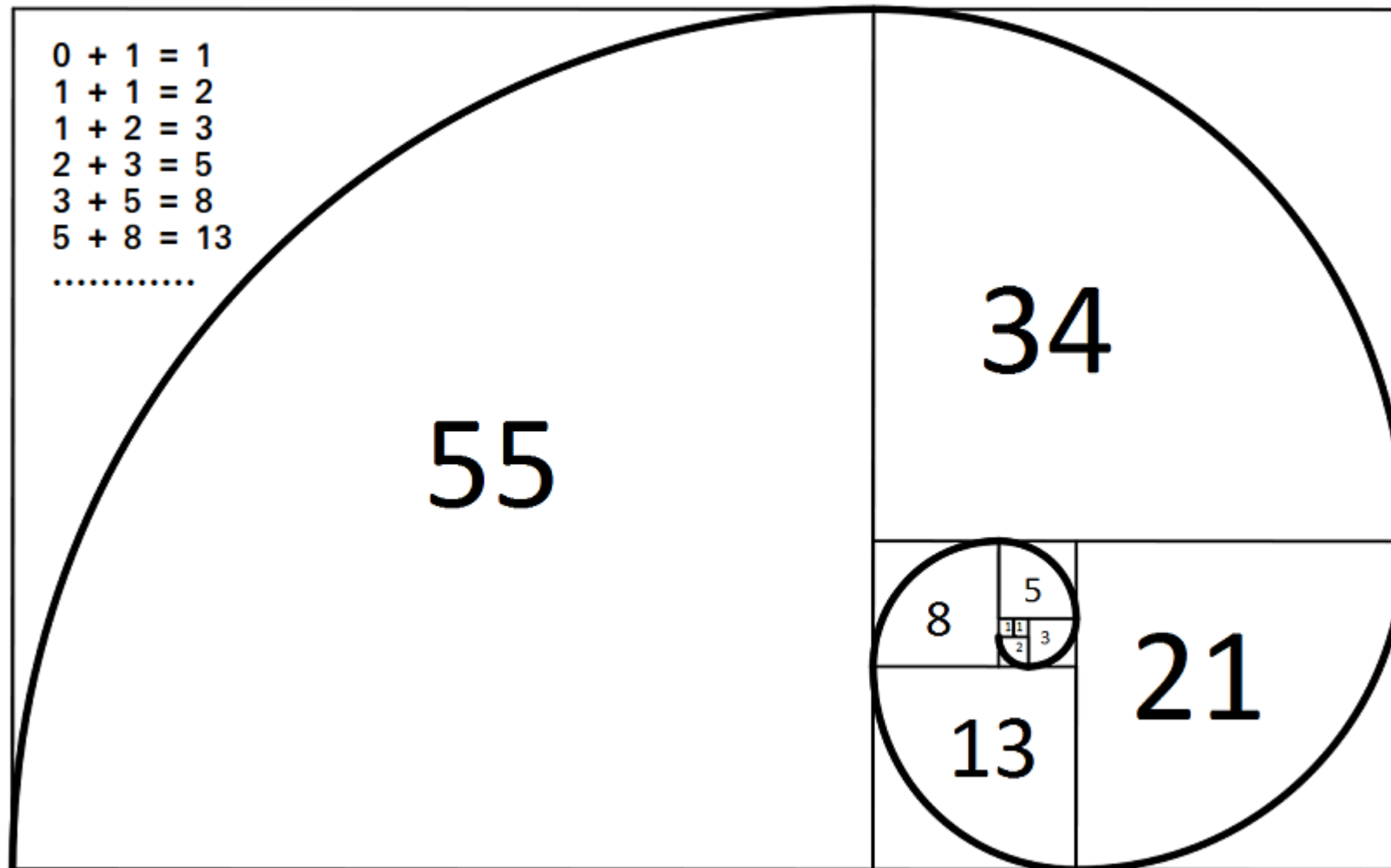
```

Теперь класс стал намного проще, а `foreach` работает так же, как и раньше.

## Пример ленивой коллекции

`yield return` позволяет создавать **ленивые последовательности** — т.е. элементы вычисляются **по запросу**, только когда они реально нужны.

Например, нам нужно вывести бесконечную последовательность чисел Фибоначчи, которые задаются очень простым правилом: каждое следующее число равно сумме двух предыдущих, а первые два числа это единицы: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...



## Пример

```
IEnumerable<int> Fibonacci()
{
    int a = 0, b = 1;
    while (true)
    {
        yield return a;
        int temp = a + b;
        a = b;
        b = temp;
    }
}
```

Использование:

```
foreach (var num in Fibonacci())
{
    if (num > 1000) break;
    Console.WriteLine(num);
}
```

## 5. Метод Equals

Когда мы работаем с объектами в C#, рано или поздно возникает вопрос: *как определить, равны ли два объекта?* Для этого в базовом классе `object` существует виртуальный метод `Equals`.

По умолчанию **для классов (ссылочных типов)** сравниваются **ссылки** на объект, т.е. равны ли они в памяти;

```
...public partial class Object
{
    ...public Object()
    {
    }

    ...~Object()
    {
    }

#pragma warning restore CA1821

    ...public virtual string? ToString()
    ...public virtual bool Equals(object? obj)
    public static bool Equals(object? objA, object? objB)
    ...public static bool ReferenceEquals(object? objA, object? objB)
    ...public virtual int GetHashCode()
}
```



Пример для класса:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}
var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };
Console.WriteLine(point1.Equals(point1)); // true (одна и та же ссылка)
Console.WriteLine(point1.Equals(point2)); // false (ссылки разные)
```

Чтобы изменить это поведение, метод можно переопределить:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public override bool Equals(object obj)
    {
        if (obj is Point point)
            return X == point.X && Y == point.Y;
        return false;
    }
}
var p1 = new Point { X = 1, Y = 1 };
var p2 = new Point { X = 1, Y = 1 };
Console.WriteLine(p1.Equals(p2)); // true (теперь сравниваются значения)
```

## Пример

Допустим, мы пишем свой список `MyList<T>` и хотим добавить метод `Contains`, чтобы проверить наличие элемента. Первая идея — использовать оператор `==`:

```
public bool Contains(T value)
{
    for (int i = 0; i < count; i++)
        if (collection[i] == value) return true; // ❌ не всегда работает
    return false;
}
```

Но тут есть проблема:

- оператор `==` определён **не для всех типов**.
- даже если сделать приведение к `object`, сравнение будет по ссылке, а это не всегда то, что нужно.

Например:

```
var list = new MyList<Point>();
list.Add(p1);
list.Add(p2);

var p3 = new Point { X = 1, Y = 1 };

Console.WriteLine(list.Contains(p3)); // ожидаем true, но может вернуть false
```

## Решение

Так как `Equals` определён у всех объектов, и его можно переопределять под нужные задачи, то правильный вариант выглядит так:

```
public bool Contains(T value)
{
    for (int i = 0; i < count; i++)
        if (collection[i].Equals(value)) return true;
    return false;
}
```

Теперь список будет корректно работать и с числами, и с классами (если они правильно переопределили `Equals`).

```
var list = new MyList<Point>();
list.Add(p1);
list.Add(p2);

var p3 = new Point { X = 1, Y = 1 };

Console.WriteLine(list.Contains(p3)); // получим true, как и ожидали
```

# Практическое задание

## Что нужно сделать:

### 1. Обновите класс `TodoList` :

- Задачи должны храниться в списке, а не в массиве.
- Поле со списком должно быть **приватным**.
- Удалите метод для увеличения массива ( `IncreaseArray` ), он больше не нужен.
- Реализуйте **индексатор** для доступа к задачам по индексу:
- Реализуйте метод-итератор с использованием `yield return` , чтобы можно было перебирать задачи через `foreach` .

### 2. Расширьте статусы задач:

- В классе `TodoItem` замените булево поле `IsDone` на свойство `Status` с типом перечисления ( `enum` ):

```
public enum TodoStatus
{
    NotStarted,    // не начато
    InProgress,    // в процессе
    Completed,     // выполнено
    Postponed,     // отложено
    Failed         // провалено
}
```

### 3. Измените команды:

- Создайте команду `StatusCommand` , которая позволяет изменять статус задачи.
- Пример ввода:

```
status 2 inprogress
status 1 completed
```

- В `CommandParser` добавьте разбор команды `status` , в `Parse` преобразуйте строку в `enum` .
- Убедитесь, что при работе программы корректно загружаются и сохраняются новые статусы.

#### 4. Измените все связанные классы и методы:

- В `TodoList` замените метод `Done(int index)` на метод `SetStatus(int index, TodoStatus status)`.
- В `ViewCommand` обновите вывод списка задач так, чтобы вместо `true/false` отображалось текстовое значение статуса.
- Добавьте описание команды `status` в команде `help`.
- В `FileManager` при сохранении и загрузке CSV вместо булевого значения используйте `status.ToString()` и `Enum.Parse<TodoStatus>()`.
- Удалите все, что было связано со старой командой `done`.

5. Делайте коммиты после **каждого изменения**. Один большой коммит будет оцениваться в два раза ниже.

6. Обновите **README.md** — добавьте описание новых возможностей программы.

7. Сделайте push изменений в GitHub.