



## Блок 4. Расширенный C#








### 4.2 — Структуры данных. Коллекции

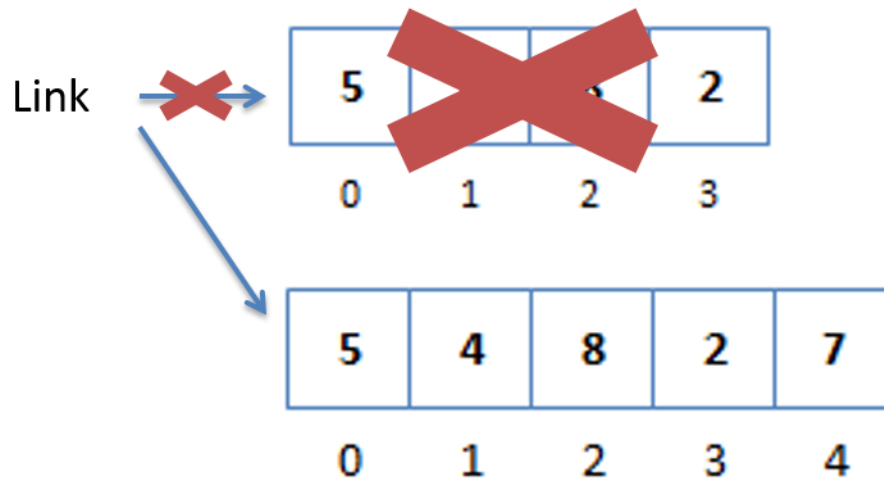
# План занятия

- Структуры данных
- Интерфейсы коллекций
- Коллекции



# Критерии оценки

-  Затраты памяти
-  Время произвольного доступа
-  Время последовательного доступа
-  Скорость поиска
-  Скорость упорядочивания
-  Скорость добавления/удаления в начало/конец
-  Скорость добавления/удаления в произвольной позиции



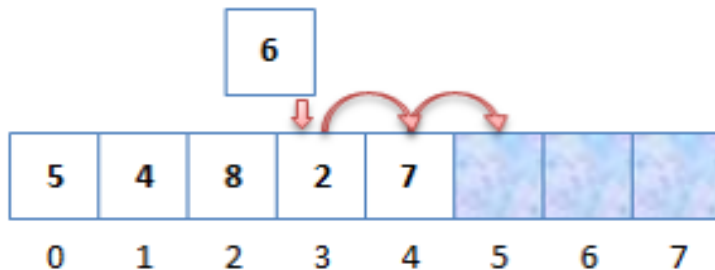
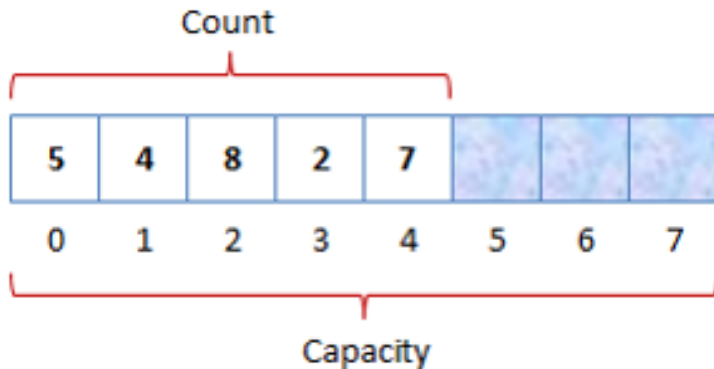
## STRENGTHS

- Память
- Произвольный доступ
- Последовательный доступ

## WEAKNESSES

- Упорядочивание
- Добавление и удаление элементов

# Динамический массив (массив с запасом)



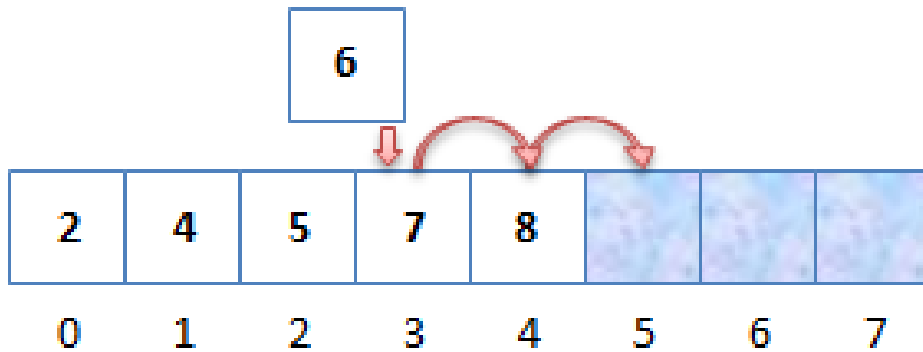
## STRENGTHS

- Произвольный доступ
- Последовательный доступ
- Добавление и удаление в конце

## WEAKNESSES

- Упорядочивание
- Добавление и удаление в начале
- Добавление и удаление в произвольной позиции

# Сортированный массив



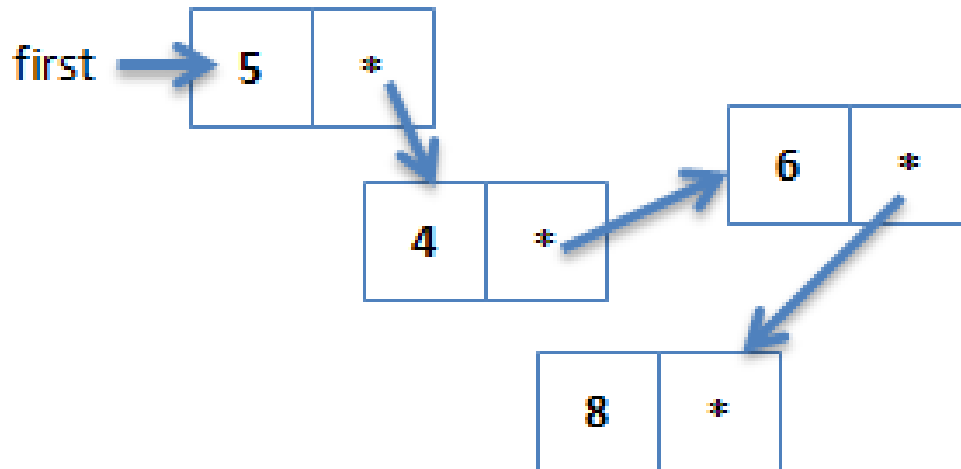
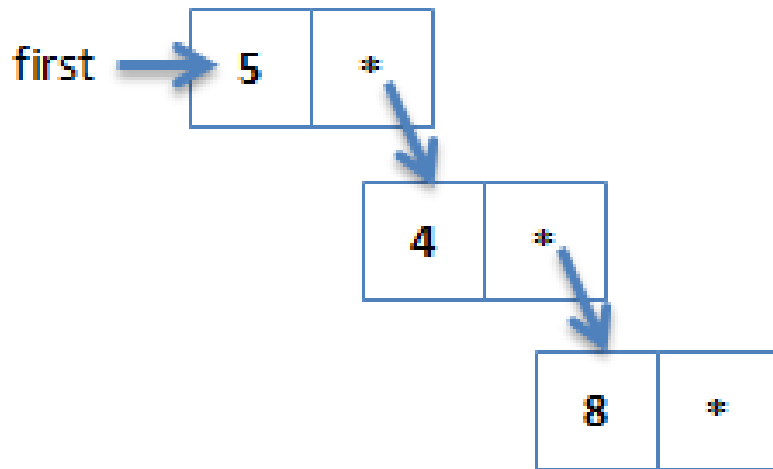
## STRENGTHS

- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание

## WEAKNESSES

- Добавление и удаление элементов

# СВЯЗНЫЙ СПИСОК



## STRENGTHS

- Последовательный доступ
- Добавление и удаление элементов

## WEAKNESSES

- Память
- Произвольный доступ
- Поиск

Иванов	данные
Петров	данные
Сидоров	данные

## STRENGTHS

---

- Произвольный доступ
- Добавление и удаление элементов

## WEAKNESSES

---

- Память
- Последовательный доступ
- Поиск



Хеширование (hashing), хеш-функция, функция свёртки

*— преобразование входного набора данных произвольной длины в выходное значение фиксированной длины.*

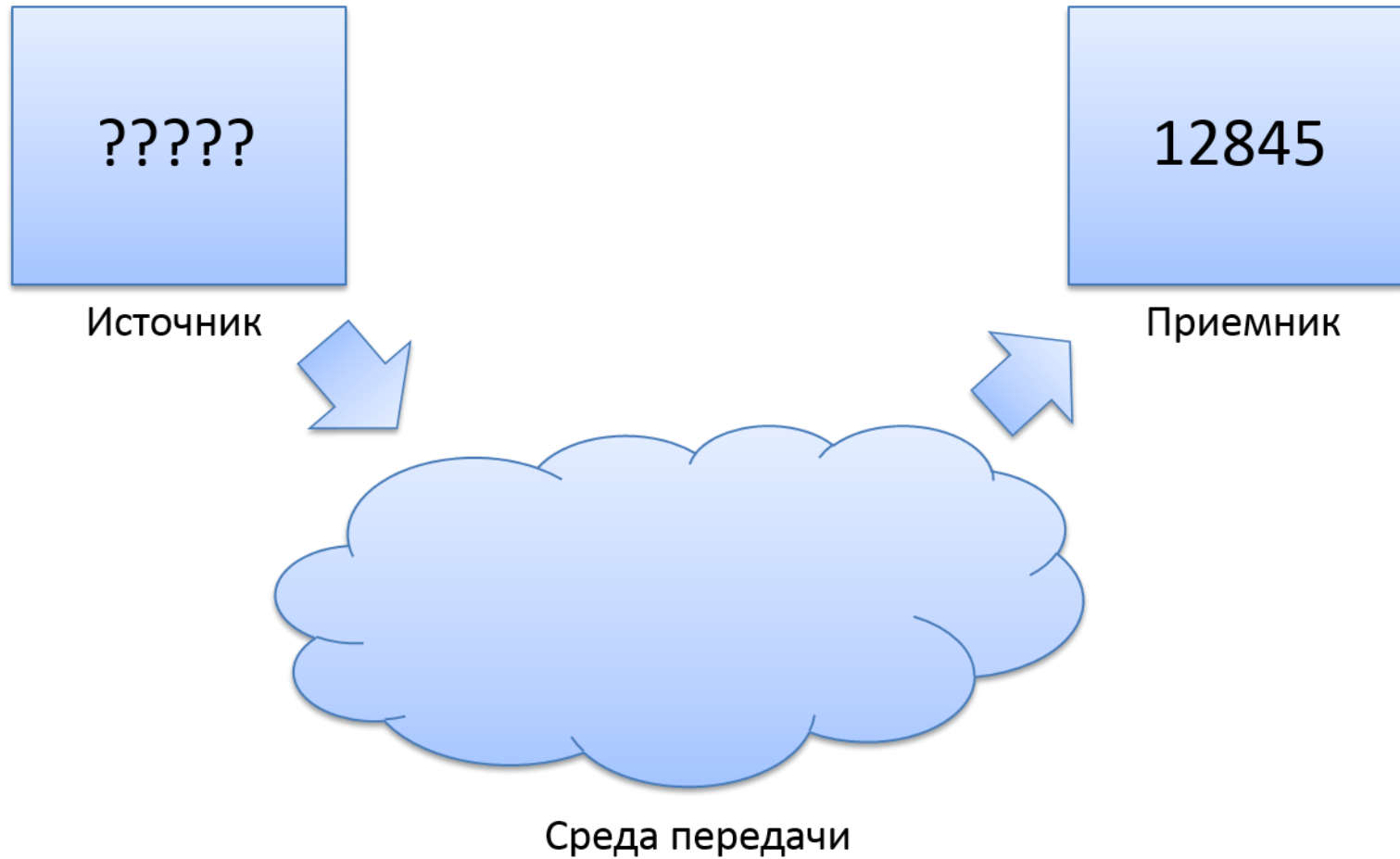
Хеш, хеш-код, дайджест сообщения (англ. message digest)

*— результат хеширования.*

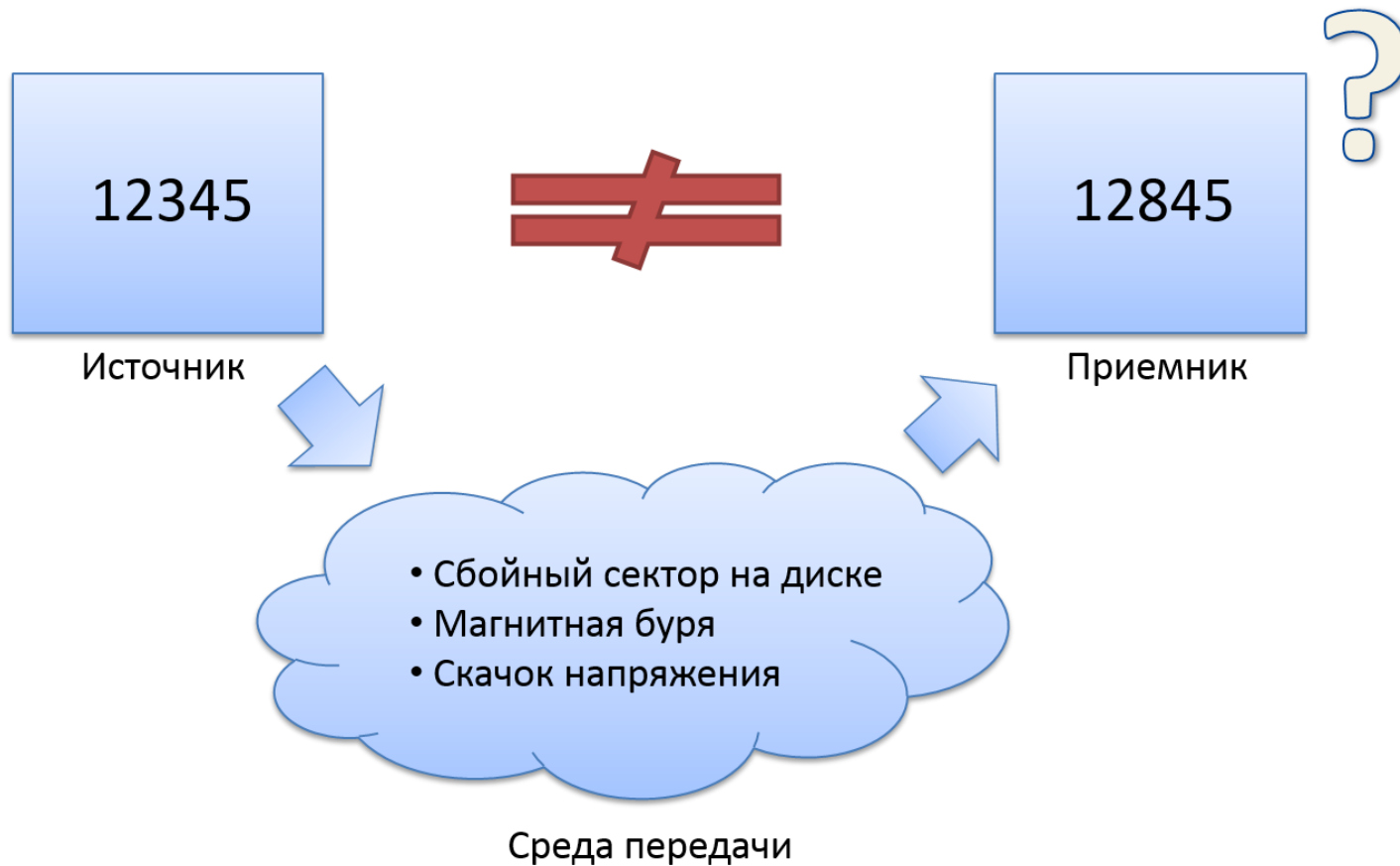
# Использование хеширования

1. Проверка целостности данных  
(контрольные суммы)
2. Проверка паролей
3. Быстрый поиск данных по ключу
  - a) Базы данных
  - b) Ассоциативные массивы

# Контрольные суммы



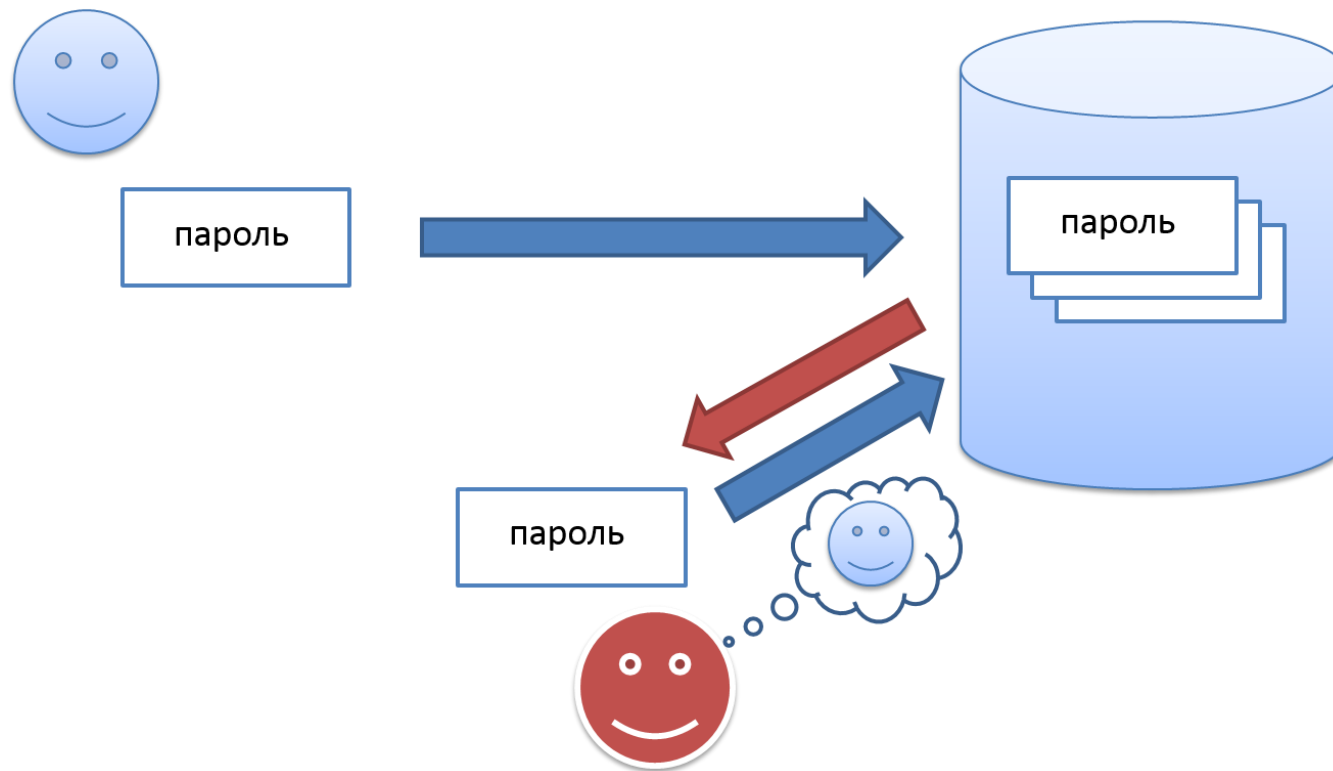
# Контрольные суммы



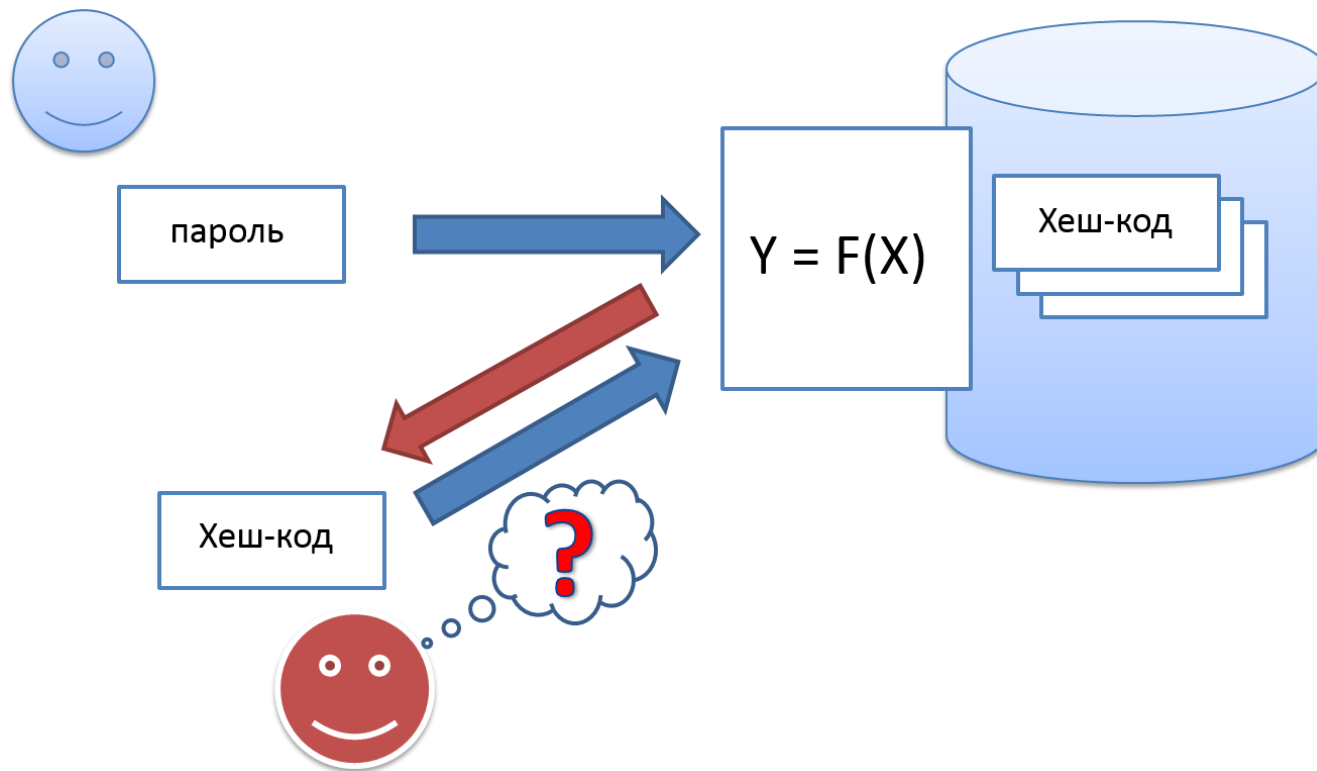
# Контрольные суммы



# Проверка паролей



# Проверка паролей



# Сравнение

Способ хранения	Время доступа	Время поиска	Добавление в конец	Добавление в произвольную позицию	Затраты памяти
Массив	const	$\sim N$	N/A	N/A	N
Массив с запасом	const	$\sim N$	const	$\sim N$	Ближайшая степень 2, не меньшая, чем N
Упорядоченный массив	const	$\sim \log_2(N)$	N/A	N/A	N
Список	$\sim N$	$\sim N$	const	const	$N + N * \text{размер ссылки}$
Хеш-таблица	const	const	N/A	N/A	Ближайшее простое число, не меньшее, чем степень 2, не меньшая, чем $1.39 * N$



- Структура данных, обеспечивающая хранение множества однотипных элементов и эффективный доступ к ним.
- Способ добавления, хранения и последующего доступа к элементам зависит от поставленной задачи.
- Существуют различные коллекции, обеспечивающие эффективность в различных классах задач.

# Основные интерфейсы коллекций

- Перечисление: `IEnumerable` и `IEnumerable<T>`
  - Коллекция: `ICollection` и `ICollection<T>`
  - Список: `IList` и `IList<T>`
  - Множество: `ISet<T>`
  - Словарь: `IDictionary` и `IDictionary<TKey, TValue>`
- 
- Обобщённые интерфейсы располагаются в пространстве имён `System.Collections.Generic`, а необобщённые — `System.Collections`.

- Предоставляет возможность перебирать элементы какой-либо последовательности.

```
public interface IEnumerator
{
    // Получение текущего элемента
    object Current { get; }

    // Переход к следующему элементу
    bool MoveNext();

    // Сброс к начальному положению
    void Reset();
}

public interface IEnumerator<T> : IEnumerator
{
    // Получение текущего элемента
    T Current { get; }
}
```

# Перечисление

- Базовый интерфейс для последовательности элементов.
- Гарантирует возможность их перебора.

```
public interface IEnumerable
{
    // Получение (создание) перечислителя
    IEnumerator GetEnumerator();
}

public interface IEnumerable<out T> : IEnumerable
{
    // Получение (создание) перечислителя
    IEnumerator<T> GetEnumerator();
}
```

- Базовый интерфейс для всех коллекций.

```
public interface ICollection : IEnumerable
{
    // Количество элементов в коллекции
    int Count { get; }

    // Копирование всех элементов коллекции в массив
    void CopyTo(Array array, int index);
}
```

# Обобщённая коллекция

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    // Количество элементов в коллекции
    int Count { get; }

    // Добавление элемента
    void Add(T item);

    // Очистка коллекции
    void Clear();

    // Проверка наличия элемента
    bool Contains(T item);

    // Копирование всех элементов коллекции в массив
    void CopyTo(T[] array, int arrayIndex);

    // Удаление элемента
    bool Remove(T item);
}
```

- Обеспечивает доступ по индексу

```
public interface IList : ICollection, IEnumerable
{
    // Добавление элемента
    int Add(object value);

    // Очистка списка
    void Clear();

    // Проверка наличия элемента
    bool Contains(object value);

    // Удаление элемента
    void Remove(object value);

    // Произвольный доступ по индексу
    object this[int index] { get; set; }

    // Поиск элемента
    int IndexOf(object value);

    // Вставка элемента в произвольную позицию
    void Insert(int index, object value);

    // Удаление элемента по индексу
    void RemoveAt(int index);
}
```

# Обобщённый список

```
public interface IList<T> : ICollection<T>, IEnumerable<T>,
                           IEnumerable
{
    // Произвольный доступ по индексу
    T this[int index] { get; set; }

    // Поиск элемента
    int IndexOf(T item);

    // Вставка элемента
    void Insert(int index, T item);

    // Удаление элемента по индексу
    void RemoveAt(int index);
}
```



- Содержит набор уникальных элементов

```
public interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    // Добавление элемента
    bool Add(T item);

    // Вычитание множества
    void ExceptWith(IEnumerable<T> other);

    // Пересечение множеств
    void IntersectWith(IEnumerable<T> other);

    // Оставляет только элементы, уникальные для обоих множеств
    void SymmetricExceptWith(IEnumerable<T> other);

    // Объединение множеств
    void UnionWith(IEnumerable<T> other);
}
```



# Множество



```
// Является строгим подмножеством
bool IsProperSubsetOf(IEnumerable<T> other);

// Является строгим надмножеством
bool IsProperSupersetOf(IEnumerable<T> other);

// Является подмножеством
bool IsSubsetOf(IEnumerable<T> other);

// Является надмножеством
bool IsSupersetOf(IEnumerable<T> other);

// Содержит хотя бы один общий элемент
bool Overlaps(IEnumerable<T> other);

// Множества эквивалентны
bool SetEquals(IEnumerable<T> other);
}
```

# Словарь пар ключ—значение

```
public interface IDictionary : ICollection, IEnumerable
{
    // Коллекция ключей
    ICollection Keys { get; }

    // Коллекция значений
    ICollection Values { get; }

    // Произвольный доступ по ключу
    object this[object key] { get; set; }

    // Добавление пары ключ—значение
    void Add(object key, object value);

    // Очистка словаря
    void Clear();

    // Проверка наличия ключа
    bool Contains(object key);

    // Удаление ключа
    void Remove(object key);
}
```

# Обобщённый словарь пар ключ—значение

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
                                           IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
{
    // Коллекция ключей
    ICollection<TKey> Keys { get; }

    // Коллекция значений
    ICollection<TValue> Values { get; }

    // Произвольный доступ по ключу
    TValue this[TKey key] { get; set; }

    // Добавление пары ключ—значение
    void Add(TKey key, TValue value);

    // Проверка наличия ключа
    bool ContainsKey(TKey key);

    // Удаление ключа
    bool Remove(TKey key);

    // Безопасное получение значения
    bool TryGetValue(TKey key, out TValue value);
}
```

- Хранят значения в переменных типа `object`.
- При извлечении элемента требуется выполнять явное приведение типа.
- При помещении объекта значимого типа автоматически выполняется упаковка.
- Проблематично контролировать соответствие добавляемых объектов предполагаемому типу.

# Обобщённые коллекции

- Хранят значения в переменных указанного типа.
- При извлечении элемента приведение типа выполнять не требуется.
- Объекты значимого типа хранятся в неупакованном виде.
- Контроль типов осуществляется компилятором.

# Зачем использовать обобщённые коллекции?

- Чтобы не выполнять постоянное приведение типа при чтении элементов.
- Чтобы не упаковывать значимые типы.
- Для дополнительной типобезопасности.
- Иначе говоря, в 99% случаев **следует использовать обобщённые коллекции.**

# Основные обобщённые коллекции языка C#

- `List<T>` — динамический массив
- `LinkedList<T>` — двунаправленный связный список
- `Queue<T>` — очередь
- `Stack<T>` — стек
- `HashSet<T>`, `SortedSet<T>` — множества
- `Dictionary<T>`, `SortedList<T>`, `SortedDictionary<T>` — словари пар ключ—значение



- Динамический массив

```
public class List<T> : IList<T>, IList,  
    ICollection<T>, ICollection,  
    IEnumerable<T>, IEnumerable
```

# LinkedList<T>

```
public class LinkedList<T> : ICollection<T>, IEnumerable<T>, ICollection, IEnumerable
{
    public LinkedListNode<T> First { get; }
    public LinkedListNode<T> Last { get; }

    public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode);
    public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);

    public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode);
    public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);

    public void AddFirst(LinkedListNode<T> node);
    public LinkedListNode<T> AddFirst(T value);

    public void AddLast(LinkedListNode<T> node);
    public LinkedListNode<T> AddLast(T value);

    public LinkedListNode<T> Find(T value);
    public LinkedListNode<T> FindLast(T value);

    public void Remove(LinkedListNode<T> node);
    public bool Remove(T value);
    public void RemoveFirst();
    public void RemoveLast();
}
```

- Очередь. Реализует принцип FIFO

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    // Извлечение с начала очереди
    public T Dequeue();

    // Помещение в конец очереди
    public void Enqueue(T item);

    // Просмотр начала очереди
    public T Peek();
}
```

- Стек. Реализует принцип LIFO

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    // Просмотр вершины стека
    public T Peek();

    // Извлечение с вершины стека
    public T Pop();

    // Помещение на вершину стека
    public void Push(T item);
}
```

- Множества

```
public class HashSet<T> : ISet<T>, ICollection<T>,
                          IEnumerable<T>, IEnumerable
```

```
public class SortedSet<T> : ISet<T>, ICollection<T>,
                             IEnumerable<T>, ICollection, IEnumerable
```

- Словари пар ключ—значение
- SortedList — упорядоченный массив пар ключ—значение
- SortedDictionary — бинарное дерево

# Спасибо за внимание!

Контактная информация:

**Дмитрий Верескун**

Инструктор

EPAM Systems, Inc.

Адрес: Саратов, Рахова, 181

Email: [Dmitry\\_Vereskun@epam.com](mailto:Dmitry_Vereskun@epam.com)

<http://www.epam.com>