Урок 5

Структуры данных. Linq выражения

Списки. Обобщенные списки. Лямбда-выражения. Linq.

Generic. Collection. Необобщенные коллекции

Обобщенные коллекции

- Список обобщенных коллекций
- Класс List<T>
- Класс HashSet<T> и SortedSet<T>
- Класс ObservableCollection<T>
- Класс LinkedList<T>
- Структура KeyValuePair<TKey, TValue>
- Класс Dictionary<TKey, TValue>
- Класс Queue<T>
- Класс Stack<T>
- Изменение порядка элементов массива на обратный
- Извлечение уникальных элементов из коллекции

Лямбда-выражения

- yield
- params
- Метод расширений

Linq

- Простой Linq-запрос
- Два where (условия)
- Еще один пример с where
- Демонстрация OrderBy и преобразования в список
- Демонстрация использования Linq с массивом пользовательских данных
- Основные Linq

Практическое задание

Дополнительные материалы

Используемая литература

На этом уроке:

1. Вы узнаете что такое структура данных
2. Научимся использовать Linq выражения
3. Познакомитесь с основными алгоритмами
4. Разберем методы расширения
5. Познакомитесь с фишками языка C#

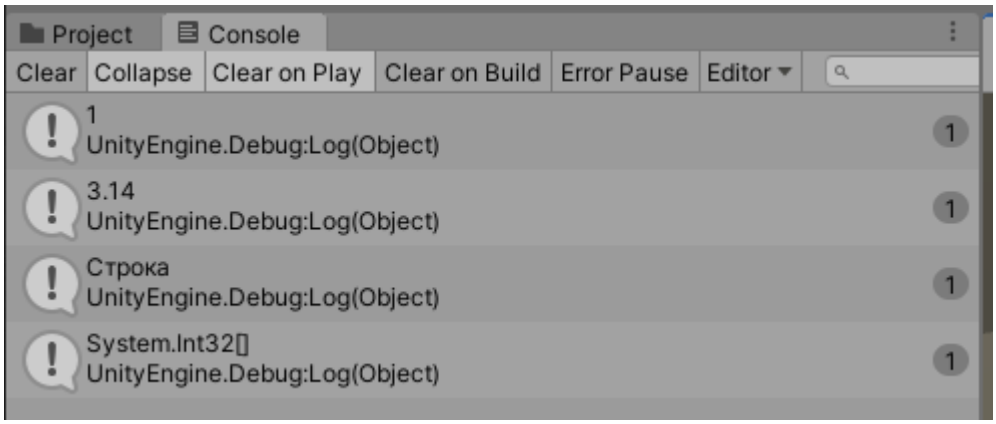
Generic. Collection. Необобщенные коллекции

Необобщенные коллекции вошли в состав среды .NET Framework еще в версии 1.0. Они определяются в пространстве имен **System.Collections**. Необобщенные коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Они позволяют манипулировать объектом любого типа, хотя и не типизированным способом. В этом их преимущество и недостаток одновременно. Благодаря тому, что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. Это удобно в тех случаях, когда требуется манипулировать совокупностью разнотипных объектов, или же когда типы хранящихся в коллекции объектов заранее не известны. Но если коллекция предназначается для хранения объекта конкретного типа, необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Пример:

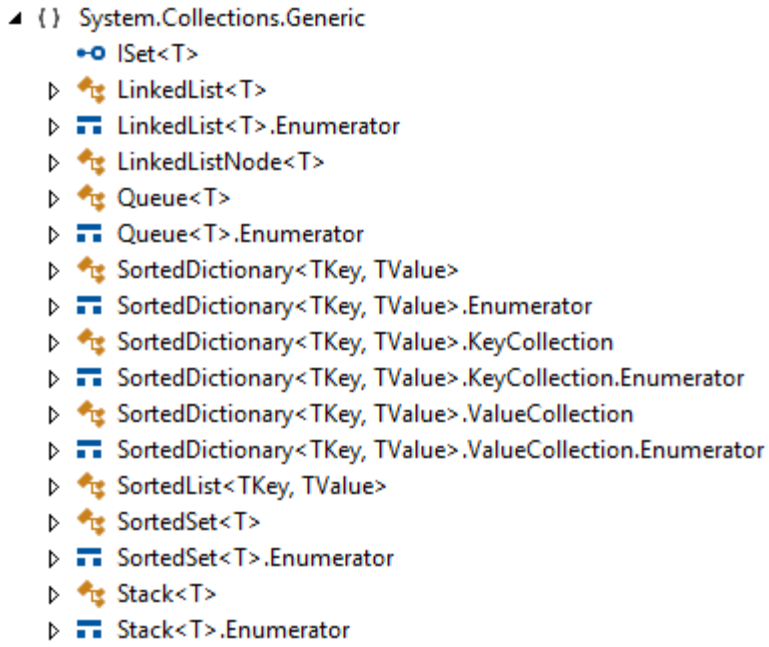
```
using System.Collections;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            ArrayList list = new ArrayList();
            list.Add(1);
            list.Add(3.14);
            list.Add("Строка");
            list.Add(new int[] { 1,2,3});
            foreach (object element in list)
            {
                Debug.Log(element);
            }
        }
    }
}
```

Вывод программы:



Обобщенные коллекции

Обобщенные объекты .Net Framework легко отличить по угловым скобкам после их названия:



В таблице описаны основные обобщенные интерфейсы, с которыми придется иметь дело при работе с обобщенными классами коллекций:

Назначение	**Интерфейс System. Collections.Generic**
Определяет общие характеристики (размер, перечисление и безопасность к потокам и др.) для всех типов обобщенных коллекций	ICollection<T>
Определяет способ сравнения объектов	IComparer<T>
Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар «ключ/значение»	IDictionary<TKey, TValue>

Возвращает интерфейс **IEnumerator<T>** для заданного объекта	IEnumerable<T>
Позволяет выполнять итерацию в стиле **foreach** по элементам коллекции	IEnumerator<T>
Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов	IList<T>
Предоставляет базовый интерфейс для абстракции множеств	ISet<T>

Список обобщенных коллекций

Назначение	Поддерживаемые основные интерфейсы	Обобщенный класс
Представляет обобщенную коллекцию ключей и значений	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	Dictionary<TKey, TValue>
Представляет двусвязный список	ICollection<T>, IEnumerable<T>	LinkedList<T>
Последовательный список элементов с динамически изменяемым размером	ICollection<T>, IEnumerable<T>, IList<T>	List<T>
Обобщенная реализация очереди – списка, работающего по алгоритму «первый вошел – первый вышел» (FIFO)	ICollection, IEnumerable<T>	Queue<T>
Обобщенная реализация словаря – отсортированного множества пар «ключ/ значение»	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	SortedDictionary<TKey, TValue>
Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дублирования	ICollection<T>, IEnumerable<T>, ISet<T>	SortedSet<T>
Обобщенная реализация стека – списка, работающего по алгоритму «последний вошел – первый вышел» (LIFO)	ICollection, IEnumerable<T>	Stack<T>
Предоставляет высокопроизводительные операции с наборами. Набор — это коллекция, которая не содержит повторяющихся элементов, элементы которой не имеют определенного порядка.	ICollection<T> IEnumerable<T> IReadOnlyCollection<T> ISet<T> IEnumerable IDeserializationCallback ISerializable	HashSet<T>
По функциональности похож на список List за тем исключением, что позволяет известить внешние объекты о том, что коллекция была изменена.	INotifyCollectionChanged INotifyPropertyChanged	ObservableCollection<T>
Экземпляр ReadOnlyCollection<T> универсального класса всегда доступен только для чтения. Коллекция, доступная только для чтения, — это просто	ICollection<T> IEnumerable<T> IList<T> IReadOnlyCollection<T> IReadOnlyList<T> ICollection IEnumerable IList	ReadOnlyCollection<T>

коллекция с оболочкой, которая предотвращает изменение коллекции		

В данной таблице приведен не полный список структур данных

Класс List<T>

В классе **List<T>** реализуется обобщенный динамический массив. Он ничем принципиально не отличается от класса необобщенной коллекции **ArrayList**.

В этом классе реализуются интерфейсы **ICollection**, **ICollection<T>**, **IList**, **IList<T>**, **IEnumerable** и **IEnumerable<T>**. У класса **List<T>** имеются следующие конструкторы:

```
public List();
public List(IEnumerable<T> collection);
public List(int capacity);
```

Первый конструктор создает пустую коллекцию класса **List** с выбираемой по умолчанию первоначальной емкостью. Второй – создает коллекцию типа **List** с количеством инициализируемых элементов, которое определяется параметром **collection** и равно первоначальной емкости массива. Третий конструктор создает коллекцию типа **List**, имеющую первоначальную емкость, задаваемую параметром **capacity**. В данном случае емкость обозначает размер базового массива, используемого для хранения элементов коллекции. Емкость коллекции, создаваемой в виде динамического массива, может увеличиваться автоматически по мере добавления в нее элементов.

Среди методов коллекции **List<T>** можно выделить следующие:

- **void Add(T item)** – добавление нового элемента в список;
- **void AddRange(ICollection collection)** – добавление в список коллекции или массива;
- **int BinarySearch(T item)** – бинарный поиск элемента в списке. Если элемент найден, метод возвращает его индекс в коллекции. При этом список должен быть отсортирован;
- **int IndexOf(T item)** – возвращает индекс первого вхождения элемента в списке;
- **void Insert(int index, T item)** – вставляет элемент **item** в списке на позицию **index**;
- **bool Remove(T item)** – удаляет элемент **item** из списка. Если удаление прошло успешно, возвращает **true**;
- **void RemoveAt(int index)** – удаляет элемент по указанному индексу **index**;
- **void Sort()** – сортировка списка.

Класс HashSet<T> и SortedSet<T>

Коллекция, содержащая только отличающиеся элементы, называется множеством (set). В составе .NET 4 имеются два множества — HashSet<T> и SortedSet<T>. Оба они реализуют интерфейс ISet<T>. Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены. При использовании данных контейнерах будет весомый прирост производительности на больших коллекциях, так как некоторые операции выполняются быстрее (например Contains, Remove, Add выполняются за O(1)).

Класс ObservableCollection<T>

Кроме стандартных классов коллекций типа списков, очередей, словарей, стеков .NET также предоставляет специальный класс ObservableCollection. Он по функциональности похож на список List за тем исключением, что позволяет известить внешние объекты о том, что коллекция была изменена.

```
using System;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleObservableCollection
    {
        private sealed class User
        {
            public string FirstName { get; }
            public string LastName { get; }

            public User(string firstName, string lastName)
            {
                FirstName = firstName;
                LastName = lastName;
            }
        }

        public void Main()
        {
            ObservableCollection<User> users = new ObservableCollection<User>
            {
                new User("Roman", "Muratov"),
                new User("Igor", "Ivanov"),
                new User("Bitalik", "Petrov")
            };

            users.CollectionChanged += Users_CollectionChanged;

            users.Add(new User("Lera", "Petrova"));
            users.RemoveAt(1);
            users[0] = new User("Sveta", "Ivanova");

            foreach (User user in users)
            {
               Debug.Log(user.FirstName);
            }
        }

        private void Users_CollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)
        {
            switch (e.Action)
            {
                case NotifyCollectionChangedAction.Add:
                    User newUser = (User)e.NewItems[0];
                    Debug.Log($@"Добавлен новый объект: {newUser.FirstName}");
                    break;
                case NotifyCollectionChangedAction.Remove:
```

```
                    User oldUser = (User)e.OldItems[0];
                    Debug.Log($@"Удален объект: {oldUser.FirstName}");
                    break;
                case NotifyCollectionChangedAction.Replace:
                    User replacedUser = (User)e.OldItems[0];
                    User replacingUser = (User)e.NewItems[0];
                    Debug.Log($@"Объект {replacedUser.FirstName} заменен объектом
{replacingUser.FirstName}");
                    break;
                case NotifyCollectionChangedAction.Move:
                    break;
                case NotifyCollectionChangedAction.Reset:
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
        }
    }
}
```

Класс LinkedList<T>

Класс **LinkedList<T>** представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент. Если в простом списке **List<T>** каждый элемент представляет объект типа **T**, то в **LinkedList<T>** каждый узел представляет объект класса **LinkedListNode<T>**. Этот класс имеет следующие свойства:

- **Value**: само значение узла, представленное типом **T**
- **Next**: ссылка на следующий элемент типа **LinkedListNode<T>** в списке. Если следующий элемент отсутствует, то имеет значение **null**
- **Previous**: ссылка на предыдущий элемент типа **LinkedListNode<T>** в списке. Если предыдущий элемент отсутствует, то имеет значение **null**

Используя методы класса **LinkedList<T>**, можно обращаться к различным элементам, как в конце, так и в начале списка:

- **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет узел **newNode** в список после узла **node**.
- **AddAfter(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением **value** после узла **node**.
- **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет в список узел **newNode** перед узлом **node**.
- AddBefore(LinkedListNode<T> node, T value): вставляет в список новый узел со значением **value** перед узлом **node**.
- **AddFirst(LinkedListNode<T> node)**: вставляет новый узел в начало списка
- AddFirst(T value): вставляет новый узел со значением **value** в начало списка
- **AddLast(LinkedListNode<T> node)**: вставляет новый узел в конец списка
- **AddLast(T value)**: вставляет новый узел со значением **value** в конец списка
- **RemoveFirst()**: удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
- **RemoveLast()**: удаляет последний узел из списка

Пример:

```
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleLinkedList
    {
        private sealed class User
        {
            public string FirstName { get; }
            public string LastName { get; }

            public User(string firstName, string lastName)
            {
                FirstName = firstName;
                LastName = lastName;
            }
        }

        public void Main()
        {
            LinkedList<int> numbers = new LinkedList<int>();

            numbers.AddLast(1); // вставляем узел со значением 1 на последнее место
            // так как в списке нет узлов, то последнее будет также и первым
            numbers.AddFirst(2); // вставляем узел со значением 2 на первое место
            numbers.AddAfter(numbers.Last, 3); // вставляем после последнего узла
новый узел со значением 3
            // теперь у нас список имеет следующую последовательность: 2, 1, 3
            foreach (int i in numbers)
            {
                Debug.Log(i);
            }

            LinkedList<User> persons = new LinkedList<User>();

            // добавляем persona в список и получим объект LinkedListNode<Person>, в
котором хранится имя Tom
            LinkedListNode<User> users = persons.AddLast(new User("Lera",
"Petrova"));
            persons.AddLast(new User("Igor", "Ivanov"));
            persons.AddFirst(new User("Ilya", "Petrov"));

            Debug.Log(users.Previous.Value.FirstName); // получаем узел перед томом и
его значение
            Debug.Log(users.Next.Value.LastName); // получаем узел после тома и его
значение
        }
    }
}
```

Структура KeyValuePair<TKey, TValue>

В пространстве имен System.Collections.Generic определена структура **KeyValuePair<TKey, TValue>**. Она служит для хранения ключа и его значения, применяется в классах обобщенных коллекций, в которых хранятся пары «ключ-значение» (например, как в классе **Dictionary<TKey, TValue>**). В этой структуре определяются два следующих свойства:

```
public TKey Key { get; };
public TValue Value { get; };
```

В этих свойствах хранятся ключ и значение соответствующего элемента коллекции.

Для построения объекта типа **KeyValuePair<TKey, TValue>** применяется конструктор:

```
public KeyValuePair(TKey key, TValue value)
```

* **key** обозначает ключ, а **value** – значение.

Класс Dictionary<TKey, TValue>

Класс **Dictionary<TKey, TValue>** позволяет хранить пары «ключ-значение» в коллекции, как в словаре. Значения доступны в словаре по соответствующим ключам. В этом отношении **Dictionary<TKey, TValue>** аналогичен необобщенному классу **Hashtable**.

В классе **Dictionary<TKey, TValue>** реализуются интерфейсы **IDictionary**, **IDictionary<TKey, TValue>**, **ICollection**, **ICollection<KeyValuePair<TKey, TValue>>**, **IEnumerable**, **IEnumerable<KeyValuePair<TKey, TValue>>**, **ISerializable** и **IDeserializationCallback**. В двух последних интерфейсах поддерживается сериализация списка. Словари имеют динамический характер, расширяясь по мере необходимости.

В классе **Dictionary<TKey, TValue>** предоставляются многочисленные конструкторы. Наиболее часто используемые:

```
public Dictionary();
public Dictionary(IDictionary<TKey, TValue> dictionary);
public Dictionary(int capacity);
```

В первом конструкторе создается пустой словарь с выбираемой по умолчанию первоначальной емкостью. Во втором – создается словарь с указанным количеством элементов **dictionary**. А в третьем конструкторе с помощью параметра **capacity** указывается емкость коллекции, создаваемой в виде словаря. Если размер словаря заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера словаря во время выполнения (как правило, это требует дополнительных затрат вычислительных ресурсов).

```
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleDictionary
    {
        public void Main()
        {
            #region ExampleDictionary

            var dict = new Dictionary<char, string>();

            dict.Add('r', "Roman");
            dict.Add('i', "Iva");
            dict.Add('v', "Viktor");
```

```
        // Перебор коллекции
        foreach (KeyValuePair<char, string> user in dict)
        {
            Debug.Log($"{user.Key} - {user.Value}");
        }

        dict['i'] = "Roman"; // Изменяем элемент с ключом i
        dict['t'] = "Roman"; // Добавляем элемент с ключом t


        foreach (KeyValuePair<char, string> user in dict)
        {
            Debug.Log($"{user.Key} - {user.Value}");
        }

        dict.Remove('i'); // Удаляем элемент по ключу

        if (dict.ContainsKey('i')) // Проверяем, имеется ли элемент с ключом i
        {
            var tempUser = dict['i']; // Получаем элемент по ключу i
        }

        // Перебор ключей
        foreach (var user in dict.Keys)
        {
            Debug.Log($"{user}");
        }

        // Перебор по значениям
        foreach (var p in dict.Values)
        {
            Debug.Log(p);
        }


        #endregion

        #region C# 5

        Dictionary<int, string> dictionary = new Dictionary<int, string>
        {
            {1,"Roman" },
            {2,"Ivan" },
            {3,"Igor" },
            {4,"Vova" }
        };

        #endregion

        #region C# 6

        Dictionary<int, string> dictionary2 = new Dictionary<int, string>
        {
            [1] = "Roman",
            [2] = "Roman",
            [3] = "Roman",
            [4] = "Roman"
        };
```

```
            #endregion
        }
    }
}
```

Класс Queue<T>

Класс **Queue** представляет обычную очередь, работающую по алгоритму FIFO («первый вошел – первый вышел»). У класса **Queue** есть 3 основных метода:

- **Dequeue** – извлекает и возвращает первый элемент очереди;
- **Enqueue** – добавляет элемент в конец очереди;
- **Peek** – просто возвращает первый элемент из начала очереди без его удаления.

```
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleQueue
    {
        public void Main()
        {
            var arr = new Queue<int>(4);

            arr.Enqueue(1); // 1
            arr.Enqueue(1); // 1 1
            arr.Enqueue(5); // 1 1 5
            arr.Enqueue(2); // 1 1 5 2

            Debug.Log(arr.Peek()); // 1 1 5 2
            Debug.Log(arr.Dequeue());// 1 5 2
        }
    }
}
```

Класс Stack<T>

Класс **Stack** представляет коллекцию, которая использует алгоритм LIFO («последний вошел – первый вышел»). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке: извлекается тот элемент, который находится выше всех в стеке. В классе **Stack** можно выделить три основных метода, которые позволяют управлять элементами:

- **Push** – добавляет элемент в стек на первое место;
- **Pop** – извлекает и возвращает первый элемент из стека;
- **Peek** – просто возвращает первый элемент из стека без его удаления.

```
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
```

```
    public sealed class ExampleStack
    {
        public void Main()
        {
            var arr = new Stack<int>(4);

            arr.Push(1); // 1
            arr.Push(1); // 1 1
            arr.Push(5); // 5 1 1
            arr.Push(2); // 2 5 1 1

            Debug.Log(arr.Peek()); // 2 5 1 1
            Debug.Log(arr.Pop()); // 5 1 1
        }
    }
}
```

Изменение порядка элементов массива на обратный

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            #region Второй способ – с использованием метода расширения Reverse

            int[] array = new int[5] { 1, 2, 3, 4, 5 };
            IEnumerable<int> reversed = array.Reverse<int>();

            #endregion
        }

        #region Первый способ – самостоятельно

        private void Reverse<T>(T[] array)
        {
            int left = 0, right = array.Length - 1;
            while (left < right)
            {
                T temp = array[left];
                array[left] = array[right];
                array[right] = temp;
                left++;
                right--;
            }
        }

        #endregion
    }
}
```

Этот код будет корректно работать на любой коллекции с интерфейсом **IEnumerable<T>**. Но возвращает он не массив, а объект-итератор, который будет перебирать элементы оригинальной коллекции в обратном порядке.

Какой способ предпочесть – решаете вы, исходя из своих потребностей.

Извлечение уникальных элементов из коллекции

Задача. Есть коллекция объектов, на основе которой вы хотите сгенерировать новую, содержащую по одной копии каждого объекта.

Решение. Чтобы сгенерировать коллекцию без повторяющихся элементов, вы должны отслеживать все элементы оригинальной коллекции и добавлять в новую лишь те, которые раньше не встречались. Рассмотрим пример:

```
private ICollection<T> GetUniques<T>(ICollection<T> list)
{
    // Для отслеживания элементов используйте словарь
    Dictionary<T, bool> found = new Dictionary<T, bool> ();
    List<T> uniques = new List<T>();
    // Этот алгоритм сохраняет оригинальный порядок элементов
    foreach (T val in list)
    {
        if (!found.ContainsKey(val))
        {
            found[val] = true;
            uniques.Add(val);
        }
    }
    return uniques;
}
```

Лямбда-выражения

C# поддерживает способность обрабатывать события «встроенным образом». С использованием анонимных методов назначается блок операторов кода непосредственно событию – вместо построения отдельного метода, подлежащего вызову делегатом. Лямбда-выражения – это всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

Чтобы подготовить фундамент для изучения лямбда-выражений, создадим новое консольное приложение **SimpleLambdaExpressions**. Затем займемся методом **FindAll ()** обобщенного типа **List<T>**. Этот метод может быть вызван, когда нужно извлечь подмножество элементов из коллекции, и он имеет следующий прототип:

```
// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match);
```

Метод возвращает объект **List<T>**, представляющий подмножество данных. Единственный параметр **FindAll ()** – обобщенный делегат типа **System.Predicate<T>**. Он может указывать на любой метод, возвращающий **bool** и принимающий единственный параметр.

Когда вызывается **FindAll ()**, каждый элемент в **List<T>** передается методу, указанному объектом **Predicate<T>**. Реализация этого метода будет производить вычисления, чтобы проверить соответствие элемента данных указанному критерию и возвратить в результате **true** или **false**. Если метод вернет **true**, текущий элемент будет добавлен в **List<T>**, представляющий искомое подмножество. Прежде чем посмотреть, как лямбда-выражения упрощают работу с **FindAll ()**, решим эту задачу в длинной нотации, используя объекты делегатов непосредственно.

Добавим класс **ExamplePredicate**, который взаимодействует с **System.Predicate<T>** для обнаружения четных чисел в списке целочисленных значений **List<T>**:

```
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExamplePredicate
    {
        public void Main()
        {
            TraditionalDelegateSyntax();
        }

        private void TraditionalDelegateSyntax()
        {
            // Создать список целых чисел
            List<int> list = new List<int>();
            list.AddRange(new int[] { 20, 1, 4, 8, 9, 4, 4 });
            // Вызов FindAll() с использованием традиционного синтаксиса делегатов
            // Создаем обобщенный экземпляр обобщенного делегата, используя
встроенный делегат Predicate
            Predicate<int> predicate = new Predicate<int>(IsEvenNumber);
            // Создаем список целых чисел, используя метод FindAll, в который
передаем делегат
            List<int> evenNumbers = list.FindAll(predicate);
            Debug.Log("Здесь только четные числа:");
            foreach (int evenNumber in evenNumbers)
            {
                Debug.Log(evenNumber);
            }
        }

        // Цель для делегата Predicate<>.
        private bool IsEvenNumber(int i)
        {
            // Это четное число?
            return i % 2 == 0;
        }
    }
}
```

Хотя этот традиционный подход к работе с делегатами функционирует ожидаемым образом, метод **IsEvenNumber ()** вызывается только при очень ограниченных условиях. В частности, когда вызывается **FindAll ()**, который взваливает на нас все заботы по определению метода. Если бы вместо этого использовался анонимный метод, код стал бы существенно яснее.

Рассмотрим новый метод в классе **Program**:

```
private void AnonymousMethodSyntax()
{
    // Создать список целых чисел
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    // Теперь использовать анонимный метод
    List<int> evenNumbers = list.FindAll(delegate (int i)
        { return (i % 2) == 0; });
    // Вывод четных чисел
    Debug.Log("Здесь только четные числа:");
    foreach (int evenNumber in evenNumbers)
    {
        Debug.Log(evenNumber);
    }
}
```

Для дальнейшего упрощения вызова **FindAll ()** можно применять лямбда-выражения. Используя этот новый синтаксис, вообще не приходится иметь дело с лежащим в основе объектом делегата. Рассмотрим новый метод в классе **Program**:

```
private void LambdaExpressionSyntax()
{
    // Создать список целых чисел
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    // Теперь использовать лямбда-выражение C#
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
    // Вывод четных чисел
    Debug.Log("Здесь только четные числа:");
    foreach (int evenNumber in evenNumbers)
    {
        Debug.Log(evenNumber);
    }
}
```

Обратите внимание на странный оператор кода, передаваемый методу **FindAll ()**, который в действительности и является лямбда-выражением. В этой модификации примера вообще нет никаких следов делегата **Predicate<T>** (как и ключевого слова **delegate**). Все, что указано вместо них – это лямбда-выражение: **i => (i % 2) == 0**

Лямбда-выражения могут применяться везде, где используется анонимный метод или строго типизированный делегат (обычно в более лаконичном виде). «За кулисами» компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата **Predicate<T>**.

yield

Использование в операторе контекстного ключевого слова yield означает, что метод, оператор или метод доступа get, в котором присутствует это ключевое слово, является итератором

```
using System.Collections;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
```

```
        private void Start()
        {
            User user = new User();

            foreach (string name in user.UsersEnum)
            {
                Debug.Log(name);
            }
        }

        private class User
        {
            private string[] Names =
            {
                "Roman",
                "Ilya",
                "Igor",
                "Lera"
            };

            public IEnumerable UsersEnum
            {
                get
                {
                    for (int i = 0; i < Names.Length; i++)
                    {
                        yield return Names[i];
                    }
                }
            }
        }
    }
}
```

params

Используя ключевое слово params, мы можем передавать неопределенное количество параметров или массив. Сам параметр с ключевым словом params при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать и должен находится последним в списке параметров

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            Addition(1, 2, 3, 4, 5);

            int[] array = new int[] { 5, 4, 3, 2, 1 };
            Addition(array);

            Addition();
        }

        private void Addition(params int[] integers)
```

```
        {
            for (int i = 0; i < integers.Length; i++)
            {
                Debug.Log(integers[i]);
            }
        }
    }
}
```

Метод расширений

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed. Метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя_типа название_параметра

Напишем класс в который будем добавлять методы расширения

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;

namespace Geekbrains
{
    public static class ExampleExtensions
    {

    }
}
```

Добавление объекта в список

```
public static T AddTo<T>(this T self, ICollection<T> coll)
{
    coll.Add(self);
    return self;
}
```

Пример использования

```
private void AddToList()
{
    var list = new List<int>();
    var list2 = new List<int>();

    list.Add(1);

    1.AddTo(list).AddTo(list2);
}
```

Проверка на какое булевское значение находится в строке

```
public static bool TryBool(this string self)
{
    return Boolean.TryParse(self, out var res) && res;
}
```

Пример использования

```
private void ExampleTryBool(string value)
{
    bool res = value.TryBool();
}
```

Проверка на какое будевское значение находится в строке

```
public static bool IsOneOf<T>(this T self, params T[] elem)
{
    return elem.Contains(self);
}
```

Пример использования

```
private void ExampleIsOneOf()
{
    var arr = new[] { 1, 5, 6, 3 };

    var t = 7;

    foreach (var i in arr)
    {
        if (i == 7)
        {

        }
    }

    if (t.IsOneOf(5, 8, 6, 3))
    {

    }
}
```

Проверка на какое будевское значение находится в строке

```
public static T GetOrAddComponent<T>(this Component child) where T : Component
{
   T result = child.GetComponent<T>() ?? child.gameObject.AddComponent<T>();
   return result;
}
```

Пример использования

```
private void ExampleGetOrAddComponent()
{
    gameObject.GetComponent<Rigidbody>().GetOrAddComponent<Rigidbody>();
}
```

Позволяет склеивать 2 массива

```
public static T[] Concat<T>(this T[] x, T[] y)
{
   if (x == null) throw new ArgumentNullException("x");
```

```
    if (y == null) throw new ArgumentNullException("y");
    var oldLen = x.Length;
    Array.Resize(ref x, x.Length + y.Length);
    Array.Copy(y, 0, x, oldLen, y.Length);
    return x;
}
```

Пример использования

```
private void ExampleConcat()
{
    int[] arrOne = new[] {1, 2, 3};
    int[] arrTwo = new[] {4, 5, 6};

    arrOne.Concat(arrTwo).Concat(new []{7, 8, 9});
}
```

Выполняет глубокое копирование ссылочного объекта

```
public static T DeepCopy<T>(this T self)
{
   if (!typeof(T).IsSerializable)
   {
      throw new ArgumentException("Type must be iserializable");
   }
   if (ReferenceEquals(self, null))
   {
      return default;
   }

   var formatter = new BinaryFormatter();
   using (var stream = new MemoryStream())
   {
      formatter.Serialize(stream, self);
      stream.Seek(0, SeekOrigin.Begin);
      return (T)formatter.Deserialize(stream);
   }
}
```

Пример использования

```
private class User
{

}

private void ExampleDeepCopy()
{
    User userOne = new User();
    User userTwo = userOne.DeepCopy();
}
```

Позволяет найти ближайшую точку из массива точек к указанной точке в параметре. Легко на Transform и на любой контейнер данных.

```
public static int ReturnNearestIndex(this Vector3[] nodes, Vector3 destination)
{
   var nearestDistance = Mathf.Infinity;
   var index = 0;
   var length = nodes.Length;
   for (var i = 0; i < length; i++)
```

```
    {
        var distanceToNode = (destination + nodes[i]).sqrMagnitude;
        if (!(nearestDistance > distanceToNode)) continue;
        nearestDistance = distanceToNode;
        index = i;
    }

    return index;
}
```

Возвращает случайный элемент из списка и при этом данный элемент не должен находится в массиве, который передается в параметре.

```
public static T ReturnRandom<T>(this List<T> list, T[] itemsToExclude)
{
    var val = list[UnityEngine.Random.Range(0, list.Count)];

    while (itemsToExclude.Contains(val))
        val = list[UnityEngine.Random.Range(0, list.Count)];

    return val;
}
```

Возвращает случайный элемент из списка.

```
public static T ReturnRandom<T>(this List<T> list)
{
    var val = list[UnityEngine.Random.Range(0, list.Count)];
    return val;
}
```

Возвращает случайный элемент из диапазона.

```
public static float GetRandom(this Vector2 v)
{
    return UnityEngine.Random.Range(v.x, v.y);
}
```

Позволяет изменить значение Vector3 по одной из координат.

```
public static Vector3 MultiplyX(this Vector3 v, float val)
{
    v = new Vector3(val * v.x, v.y, v.z);
    return v;
}

public static Vector3 MultiplyY(this Vector3 v, float val)
{
    v = new Vector3(v.x, val * v.y, v.z);
    return v;
}

public static Vector3 MultiplyZ(this Vector3 v, float val)
{
    v = new Vector3(v.x, v.y, val * v.z);
    return v;
}
```

Позволяет расширить массив на заданное количество элементов.

```
public static T[] Increase<T>(this T[] values, int increment)
{
    T[] array = new T[values.Length + increment];
```

```
    values.CopyTo(array, 0);
    return array;
}
```

Позволяет найти объект по имени в иерархии.

```
public static Transform FindDeep(this Transform obj, string id)
{
    if (obj.name == id)
    {
        return obj;
    }

    var count = obj.childCount;
    for (var i = 0; i < count; ++i)
    {
        var posObj = obj.GetChild(i).FindDeep(id);
        if (posObj != null)
        {
            return posObj;
        }
    }

    return null;
}
```

Позволяет найти все компоненты вложенных объектов.

```
public static List<T> GetAll<T>(this Transform obj)
{
    var results = new List<T>();
    obj.GetComponentsInChildren(results);
    return results;
}
```

Позволяет изменить альфу у выбранного цвета.

```
public static Color SetColorAlpha(this Color c, float alpha)
{
    return new Color(c.r, c.g, c.b, alpha);
}
```

Linq

Linq использует следующие программные конструкции:

- Неявная типизация локальных переменных;
- Синтаксис инициализации объектов и коллекций;
- Лямбда-выражения;
- Расширяющие методы;
- Анонимные типы.

В основу Linq положено понятие запроса, в котором определяется информация, получаемая из источника данных. Например, запрос списка рассылки почтовых сообщений заказчикам может потребовать предоставления адресов всех заказчиков, проживающих в конкретном городе; запрос базы данных товарных запасов – список товаров, запасы которых исчерпались на складе; а запрос журнала, регистрирующего интенсивность использования Интернета, – список наиболее часто посещаемых веб-сайтов. И хотя все эти запросы отличаются в деталях, их можно выразить, используя одни и те же синтаксические элементы Linq. Как только запрос будет сформирован, его можно

выполнить. Это делается, в частности, в цикле **foreach**. В результате выполнения запроса выводятся его результаты. Поэтому использование запроса может быть разделено на две главные стадии. На первой запрос формируется, а на второй – выполняется. Таким образом, при формировании запроса определяется, что именно следует извлечь из источника данных. А при выполнении запроса выводятся конкретные результаты.

Для обращения к источнику данных по запросу, сформированному средствами Linq, в этом источнике должен быть реализован интерфейс **IEnumerable**. Он имеет две формы: обобщенную и необобщенную. Как правило, работать с источником данных легче, если в нем реализуется обобщенная форма **IEnumerable<T>**, где **T** обозначает обобщенный тип перечисляемых данных. Здесь и далее предполагается, что в источнике данных реализуется форма интерфейса **IEnumerable<T>**. Этот интерфейс объявляется в пространстве имен **System.Collections.Generic**. Класс, в котором реализуется форма интерфейса **IEnumerable<T>**, поддерживает перечисление – это означает, что его содержимое может быть получено по очереди или в определенном порядке. Форма интерфейса **IEnumerable<T>** поддерживается всеми массивами в C#. Поэтому на примере массивов можно наглядно продемонстрировать основные принципы работы Linq. Применение Linq не ограничивается массивами. Есть два способа выполнения запроса Linq: отложенное и немедленное. При отложенном выполнении Linq-выражение не выполняется, пока не будет произведена итерация или перебор по выборке.

Простой Linq-запрос

```
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            // Все массивы в C# неявным образом преобразуются в форму интерфейса
IEnumerable<T>
// Благодаря этому любой массив в C# может служить в качестве источника данных,
извлекаемых
// по запросу Linq
            int[] nums = { 1, -2, 3, 0, -4, 5 };
// Создадим запрос, получающий только положительные числа
// posNums – переменная запроса
// Ей присваивается результат выполнения запроса
            var posNums = from n       // n -переменная диапазона (как в foreach)
                    in nums  // Источник данных
                where n > 0  // Предикат (условие) – фильтр данных
                select n;      // Какие данные получаем? В сложных запросах здесь
можно указать,например, фамилию адресата вместо всего адреса
            Debug.Log("Положительные числа: ");
// Выполняем запрос и выводим положительные числа на экран
            foreach (int i in posNums)
            {
                Debug.Log(i + " ");
            }
        }
    }
}
```

Два where (условия)

```
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
// Создаем запрос на выборку положительных чисел меньше 10
            var posNums = from n in nums
                where n > 0
                where n < 10
                select n;
            Debug.Log("Положительные числа меньше 10: ");
            foreach (int i in posNums)
            {
                Debug.Log(i + " ");
            }
        }
    }
}
```

Еще один пример с where

```
using System;
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            string[] strs = { ".com", ".net", "hsNameA.com", "hsNameB.net",
                "test", ".network", "hsNameC.net", "hsNameD.com" };
// Create a query that obtains Internet addresses that end with .net
// Создадим запрос, который получает все Интернет-адреса, заканчивающиеся на .net
            var netAddrs = from addr in strs
                where addr.Length > 4
                      && addr.EndsWith(".net", StringComparison.Ordinal)
// Он  возвращает логическое значение true, если вызывающая его строка оканчивается
// последовательностью символов, указываемой в качестве аргумента этого метода
// Сортировка результатов запроса с помощью оператора orderby
                select addr;
// Выполним запрос и выведем результаты
            foreach(var str in netAddrs)
            {
                Debug.Log(str);
            }
        }
```

```
    }
}
```

Демонстрация OrderBy и преобразования в список

```
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            int[] nums = { 10, -19, 4, 7, 2, -5, 0 };
// Запрос, который получает значения в отсортированном порядке
            var posNums = from n in nums
                where n>0  orderby n ascending
                select n;
            Console.Write("Значения по возрастанию: ");
// Преобразовать в список (для примера такой возможности)
            List<int> a = posNums.ToList<int>();
// Execute the query and display the results
            foreach (int i in posNums)
            {
                Debug.Log(i + " ");
            }

            nums[1] = 10;
            Debug.Log("\n"+posNums.Sum());
        }
    }
}
```

Демонстрация использования Linq с массивом пользовательских данных

```
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        class EmailAddress
        {
            public string Name { get; }
            public string Address { get; }
            public EmailAddress(string n, string a)
            {
                Name = n;
                Address = a;
            }
        }
```

```
        private void Start()
        {
            EmailAddress[] addrs = {
                new EmailAddress("Herb", "Herb@HerbSchildt.com"),
                new EmailAddress("Tom", "Tom@HerbSchildt.com"),
                new EmailAddress("Sara", "Sara@HerbSchildt.com")
            };
            // Create a query that selects e-mail addresses
            var eAddrs = from entry in addrs
                select entry.Address;
            Debug.Log("The e-mail addresses are");
            // Execute the query and display the results
            foreach (string s in eAddrs)
            {
                Debug.Log("   " + s);
            }
        }
    }
}
```

Основные Linq

Создадим демонстрационный класс с экспериментальными данными

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleLinq
    {
        private sealed class User
        {
            public string FirstName { get; }
            public string LastName { get; }
            public int Age { get; set; }

            public User(string firstName, string lastName)
            {
                FirstName = firstName;
                LastName = lastName;
            }
        }

         private readonly User[] _users;
         private readonly int[] _numbers;

        public ExampleLinq()
        {
            _users = new []
            {
                new User("Roman", "Muratov") {Age = 18},
                new User("Ivan", "Petrov"){Age = 22},
                new User("Ilya", "Afanasyev"){Age = 18},
                new User("Igor", "Ivanov"){Age = 25},
                new User("Lera", "Muratova"){Age = 17},
```

```
                new User("Sveta", "Petrova"){Age = 27},
                new User("Lena", "Ivanova"){Age = 33},
                new User("Lera", "Muratova"){Age = 17},
                new User("Sveta", "Petrova"){Age = 27},
                new User("Lena", "Ivanova"){Age = 33}
            };
            _numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
        }
    }
}
```

Фильтрация объектов

```
public void Filtration()
{
    IEnumerable<int> evens = from i in _numbers
                             where i % 2 == 0 && i > 10
                             select i;

    IEnumerable<int> evens1 = _numbers.Where(i => i % 2 == 0 && i > 10);

    foreach (int i in evens)
    {
        Debug.Log(i);
    }
}
```

Выборка сложных объектов

```
public void SelectingComplexObjects()
{
    var selectedUsers = from user in _users
                        where user.Age > 28
                        select user;

    var selectedUsers1 = _users.Where(u => u.Age > 28).ToList();

    foreach (User user in selectedUsers)
    {
        Debug.Log($"{user.FirstName} - {user.Age}");
    }
}
```

Проекция

```
public void Projection()
{
    var names = _users.Select(u => u.FirstName);

    foreach (string user in names)
    {
        Debug.Log(user);
    }
}
```

Переменые в запросах и оператор let

```
public void ExampleLet()
{
    var people = from u in _users
                 let age = u.Age <= 18 ? u.Age + (18 - u.Age) : u.Age
                 select new User(u.FirstName, u.LastName)
```

```
                {
                    Age = age
                };

    foreach (var user in people)
    {
        Debug.Log($"{user.FirstName} - {user.Age}");
    }
}
```

Выборка из нескольких источников

```
public void SamplingFromSeveralSources()
{
    var people = from user in _users
                 from number in _numbers
                 select new { Name = user.LastName, Number = number };

    foreach (var p in people)
    {
        Debug.Log($"{p.Name} - {p.Number}");
    }
}
```

Сортировка

```
public void Sorting()
{
    var sortedUsers = from u in _users
                      orderby u.Age //ascending (сортировка по возрастанию) и
descending (сортировка по убыванию)
                      select u;

    var result = _users.OrderBy(u => u.FirstName).ThenBy(u => u.Age).ThenBy(u =>
u.FirstName.Length); //ThenByDescending() (для сортировки по убыванию)


    foreach (User u in sortedUsers)
    {
        Debug.Log($"{u.FirstName} - {u.Age}");
    }
}
```

Работа с множествами

```
public void WorkingWithSets()
{
    string[] peopleFromAstrakhan = { "Igor", "Roman", "Ivan" };
    string[] peopleFromMoscow = { "Ilya", "Vitalik", "Denis" };

    // разность множеств
    var result = peopleFromAstrakhan.Except(peopleFromMoscow);
    // пересечение множеств
    var result1 = peopleFromAstrakhan.Intersect(peopleFromMoscow);
    // объединение множеств
    var result2 = peopleFromAstrakhan.Union(peopleFromMoscow);
    // удаление дубликатов
    var result3 = peopleFromAstrakhan.Concat(peopleFromMoscow).Distinct();
}
```

Агрегатные операции

```
public void ExampleAverage()
{
    int min1 = _numbers.Min();
    int min2 = _users.Min(n => n.Age); // минимальный возраст

    int max1 = _numbers.Max();
    int max2 = _users.Max(n => n.Age); // максимальный возраст

    double avr1 = _numbers.Average();
    double avr2 = _users.Average(n => n.Age); //средний возраст
}
```

Методы Skip и Take

```
public void ExampleSkipAndTake()
{
    var result = _numbers.Take(3); // три первых элемента
    var result1 = _numbers.Skip(3); // все элементы, кроме первых трех

    foreach (var t in _users.TakeWhile(x => x.FirstName.StartsWith("I")))
    {
        Debug.Log(t);
    }

    foreach (var t in _users.SkipWhile(x => x.FirstName.StartsWith("I")))
    {
        Debug.Log(t);
    }
}
```

Группировка

```
public void ExampleGrouping()
{
    var groups = from user in _users
                 group user by user.LastName;

    foreach (IGrouping<string, User> g in groups)
    {
        Debug.Log(g.Key);
        foreach (var t in g)
        {
            Debug.Log(t.FirstName);
        }
    }

    var groups1 = _users.GroupBy(p => p.LastName)
                .Select(g => new { LastName = g.Key, Count = g.Count() });

    var groups2 = _users.GroupBy(p => p.LastName)
                .Select(g => new
                {
                    LastName = g.Key,
                    Count = g.Count(),
                    Name = g.Select(p => p)
                });

    foreach (var g in groups2)
    {
        Debug.Log(g.LastName);
        Debug.Log(g.Count);
```

```
        foreach (var t in g.Name)
        {
            Debug.Log(t.FirstName);
        }
    }
}
```

Методы All и Any

```
public void ExampleAllAndAny()
{
    bool result = _users.All(u => u.Age > 20);
    Debug.Log(result
        ? "У всех пользователей возраст больше 20"
        : "Есть пользователи с возрастом меньше 20");

    bool result1 = _users.Any(u => u.Age < 20);
    Debug.Log(result1
        ? "Есть пользователи с возрастом меньше 20"
        : "У всех пользователей возраст больше 20");
}
```

Соединение коллекций. Метод Join, GroupJoin и Zip

```
private sealed class People
{
    public string FirstName { get; }
    public string LastName { get; }
    public string Country { get; set; }

    public People(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}

public void ExampleJoin()
{
    User[] users = { new User("Roman", "Muratov") {Age = 18},
        new User("Ivan", "Petrov"){Age = 22},
        new User("Ilya", "Afanasyev"){Age = 18} };
    People[] peoples = { new People("Lera", "Muratova"){Country = "Astrakhan"},
        new People("Sveta", "Petrova"){Country = "Moscow"},
        new People("Lena", "Ivanova"){Country = "Minsk"} };

    var resultJoin = from pl in users
        join t in peoples on pl.LastName equals t.LastName
        select new { Name = pl.FirstName, Team = pl.LastName, Country = t.Country };

    foreach (var item in resultJoin)
    {
        Debug.Log($"{item.Name} - {item.Team} ({item.Country})");
    }

    resultJoin = users.Join(peoples,
        p => p.LastName,
        t => t.LastName,
        (p, t) => new { Name = p.FirstName, Team = p.LastName, Country = t.Country
});
```

```
    var resultGroupJoin = peoples.GroupJoin(
        users,
        t => t.LastName,
        pl => pl.LastName,
        (team, pls) => new
        {
            Name = team.LastName,
            Country = team.Country,
            Players = pls.Select(p=>p.FirstName)
        });

    foreach (var team in resultGroupJoin)
    {
        Debug.Log(team.Name);
        foreach (string player in team.Players)
        {
            Debug.Log(player);
        }
    }

    var resultZip = users.Zip(peoples,
        (player, team) => new
        {
            Name = player.FirstName,
            LastName = team.LastName, Country = team.Country
        });

    foreach (var player in resultZip)
    {
        Debug.Log($"{player.Name} - {player.LastName} ({player.Country})");
    }
}
```

Практическое задание

1. Расписать в текстовом документе плюсы и минусы изученных структур данных и расписать примеры в которых нужно будет использовать тот или иной контейнер
2. Реализовать метод расширения для поиска количество символов в строке
3. Дана коллекция **List<T>**. Требуется подсчитать, сколько раз каждый элемент встречается в данной коллекции:
    a. для целых чисел;
    b. * для обобщенной коллекции;
    c. ** используя Linq.
4. * Дан фрагмент программы:

```
Dictionary<string, int> dict = new Dictionary<string, int>()
{
      {"four",4 },
```

```
        {"two",2 },
        { "one",1 },
        {"three",3 },
};
var  d  =  dict.OrderBy(delegate(KeyValuePair<string,int>  pair)  {  return
pair.Value; });
foreach (var pair in d)
{
        Debug.Log($"{pair.Key} - {pair.Value}");
}
```

a. Свернуть обращение к **OrderBy** с использованием лямбда-выражения **=>.**

b. * Развернуть обращение к **OrderBy** с использованием делегата

Дополнительные материалы

1. https://freedevgame.ru/prog/50-sovetov-po-rabote-s-unity.html
2. https://habr.com/ru/post/309478/
3. https://habr.com/ru/company/badoo/blog/345710/

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Язык программирования C# 7 и платформы .NET и .NET Core | Троелсен Эндрю, Джепикс Филипп – 2018 г.
2. Unity в действии. Мультиплатформенная разработка на C# | Джозеф Хокинг - 2016 г.
3. https://docs.unity3d.com/Manual/index.html
4. MSDN