

Лекция №3. Списки и словари

Список

Мы уже много работали со стандартным списком `List`. Давайте попробуем написать собственную реализацию списка, для того, чтобы лучше разобраться как он работает и узнать ряд возможностей языка C#, которые мы сможем использовать в дальнейшем.

Вспомним главные особенности списков:

- Список — это коллекция с изменяемым числом элементов, мы можем добавлять и удалять их;
- В списке может храниться любой тип данных, поэтому он должен быть generic-типом;
- Значения в списке на самом деле хранятся в массиве фиксированной длины, когда он полностью заполняется, создается новый массив, в два раза больше предыдущего, в него копируются значения из старого массива и добавляется новое значение, которое не помещалось в старом массиве.

```
class MyList<T>
{
    //массив в котором хранятся все значения
    public T[] collection;
    //количество элементов в списке
    int count = 0;
    public MyList()
    {
        collection = new T[100];
    }
    //Метод для создания массива большего размера
    void Enlarge()
    {
        var newCollection = new T[collection.Length*2];
        for(int i = 0; i < collection.Length; i++)
            newCollection[i] = collection[i]
        collection = newCollection;
    }
    //Метод для добавления нового элемента в список
    public void Add(T value)
    {
        if (count == collection.Length)
            Enlarge();
        collection[count++] = value;
    }
}
```

```
}  
}
```

Список также является итерируемым объектом, то есть его элементы можно перебирать через цикл `foreach`. Реализуем это:

```
class MyList<T> : IEnumerable<T>  
{  
    ...  
    public IEnumerator<T> GetEnumerator()  
    {  
        for (int i = 0; i < count; i++)  
            yield return collection[i];  
    }  
    IEnumerator IEnumerable.GetEnumerator()  
    {  
        return GetEnumerator();  
    }  
}
```

Теперь мы можем перебирать элементы в списке. Но что если мы хотим получить конкретный элемент? Обычно для этого используют квадратные скобки, в которых указывается какой-то ключ или индекс, по которому можно получить конкретный элемент (например `list[1]`, получим элемент с индексом 1). Для того, чтобы добавить в наш список возможность получать элемент по индексу таким способом, нужно написать в классе такую конструкцию:

```
class MyList<T> : IEnumerable<T>  
{  
    ...  
    public T this[int index]  
    {  
        get  
        {  
            if (index < 0 || index >= count) throw new  
IndexOutOfRangeException();  
            return collection[index];  
        }  
        set  
        {  
            if (index < 0 || index >= count) throw new  
IndexOutOfRangeException();  
            collection[index] = value;  
        }  
    }  
}
```

Теперь мы можем добавлять элементы в `MyList`, перебирать их в цикле `foreach`, получать и изменять элементы по индексу:

```
var list = new MyList<int>();
list.Add(1);
list.Add(2);
list.Add(3);

foreach (var e in list)
    Console.WriteLine(e); //1 2 3

list[1] = 5;
Console.WriteLine(list[1]); //5
```

Метод Equals

Давайте добавим в наш список метод `Contains`, который будет проверять есть ли определенный элемент в списке. Эта задача выглядит не сложной, самый простой способ, который может прийти в голову выглядит так:

```
class MyList<T>
{
    ...
    public bool Contains(T value)
    {
        for (int i = 0; i < count; i++)
            //здесь будет ошибка
            // == не определено для всех типов данных
            if (collection[i] == value) return true;
        return false;
    }
}
```

Но он не будет работать. `MyList` — generic-тип, это значит, что в нем может храниться любой тип данных, но операция `==` определена не для всех типов, поэтому компилятор покажет нам ошибку. Можно конечно сделать upcast элементов массива к типу `object`, в этом случае можно будет сравнивать любые элементы.

```
class MyList<T>
{
    ...
    public bool Contains(T value)
    {
        for (int i = 0; i < count; i++)
            //не имеет смысла, потому что будут сравниваться ссылки на объекты
            if ((object)collection[i] == (object)value) return true;
    }
}
```

```

        return false;
    }
}

```

У такого способа есть проблемы. По умолчанию объекты сравниваются по ссылке, но часто нам необходимо сравнивать элементы по значению. Рассмотрим такой пример:

```

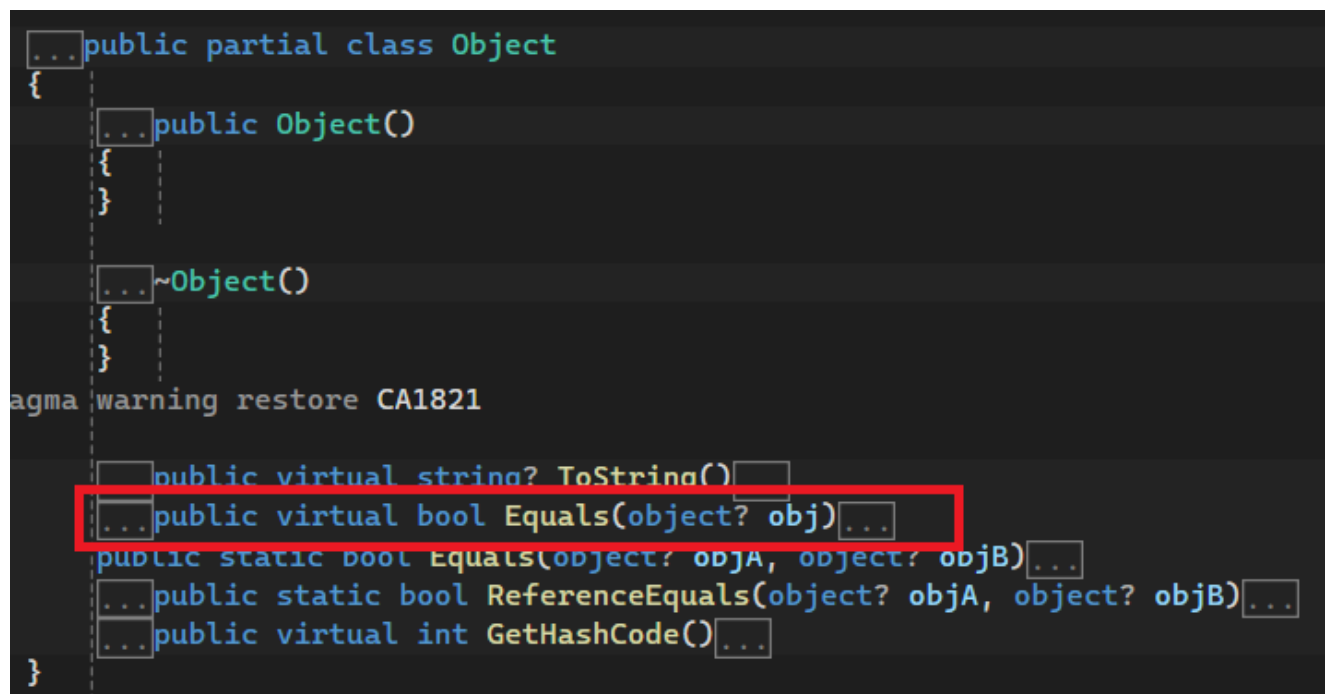
var list = new MyList<int>();
list.Add(1);
list.Add(2);
list.Add(3);

//выведет false
Console.WriteLine(list.Contains(1));

```

При вызове метода `list.Contains(1)` мы ожидаем, что он вернет `true`, т.к. в списке действительно есть 1. Но если мы запустим программу, то увидим, что он выведет `false`, из-за того, что числа будут сравниваться по ссылке, а ссылки у них разные.

Как же тогда реализовать метод `Contains`? Если мы посмотрим, какие методы есть в классе `object`, то увидим там виртуальный метод `Equals`, этот метод позволяет определить равен ли объект, который передали как аргумент, текущему объекту.



```

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

    ...~Object()
    {
    }

    ...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()
}

```

The screenshot shows the source code of the `Object` class in C#. The `Equals(object? obj)` method is highlighted with a red rectangle. The code is displayed in a dark-themed editor with syntax highlighting.

Классы по умолчанию сравниваются по ссылке, т.к. они являются ссылочными типами:

```

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 - т.к. тот же самый объект, ссылка одинаковая
Console.WriteLine(point1.Equals(point2));
//false - т.к. ссылки разные
```

А структуры по значению:

```
struct 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
Console.WriteLine(point1.Equals(point2)); //true
```

Однако мы можем переопределить этот метод так, как нам надо:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public override bool Equals(object obj)
    {
        if (!(obj is Point)) return false;
        var point = obj as Point;
        return X == point.X && Y == point.Y;
    }
}

var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };

Console.WriteLine(point1.Equals(point1)); //true
Console.WriteLine(point1.Equals(point2)); //true
```

Именно этот метод мы должны использовать в `Contains`, так как

1. `Equals` определен для всех объектов

2. Элементы будут сравниваться по разному, в зависимости от того, как у них определен метод `Equals`

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

Перегрузка операторов

Как мы уже поняли, сравнивать любые объекты можно используя метод `Equals`, однако такая запись не очень удобна, намного привычнее использовать `==` для этого. И мы можем реализовать сравнение через `==`, для этого нужно просто написать перегрузку для соответствующего оператора:

```
class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public static bool operator ==(Point p1, Point p2)
    {
        return p1.X == p2.X && p1.Y == p2.Y;
    }
}

//теперь мы можем сравнивать точки так
var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };
Console.WriteLine(point1 == point2); //true
```

Точно также мы можем перегрузить и `!=`:

```
class Point
{
    ...
    public static bool operator !=(Point p1, Point p2)
    {
        return !(p1 == p2);
    }
}
```

```

}

//теперь мы можем сравнивать точки так
var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };
Console.WriteLine(point1 == point2); //true
Console.WriteLine(point1 != point2); //false

```

И другие:

```

class Point
{
    ...
    public static Point operator +(Point p1, Point p2)
    {
        return new Point { X = p1.X + p2.X, Y = p1.Y + p2.Y };
    }

    public static Point operator -(Point p1, Point p2)
    {
        return new Point { X = p1.X - p2.X, Y = p1.Y - p2.Y };
    }

    public static double operator *(Point p1, Point p2)
    {
        return p1.X * p2.X + p1.Y * p2.Y;
    }
}

var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };

//теперь мы можем складывать, вычитать и умножать точки
point1 + point2;
point1 - point2;
point1 * point2;
//операторы +=, *= и подобные переопределяются автоматически
point += point2;

```

Кроме того можно определить операции с другими типами данных, но хотя бы один аргумент должен быть того класса, в котором он объявлен:

```

class Point
{
    ...
    public static Point operator *(Point p, double value)
    {
        return new Point { X = p.X * value, Y = p.Y * value };
    }
}

```

```

    }

    //для того чтобы можно было умножать число на точку в любом порядке
    public static Point operator *(double value, Point p)
    {
        return p * value;
    }

    //а вот так писать нельзя
    //public static value operator *(double value1, double value2) {}
}

var point1 = new Point { X = 1, Y = 1 };

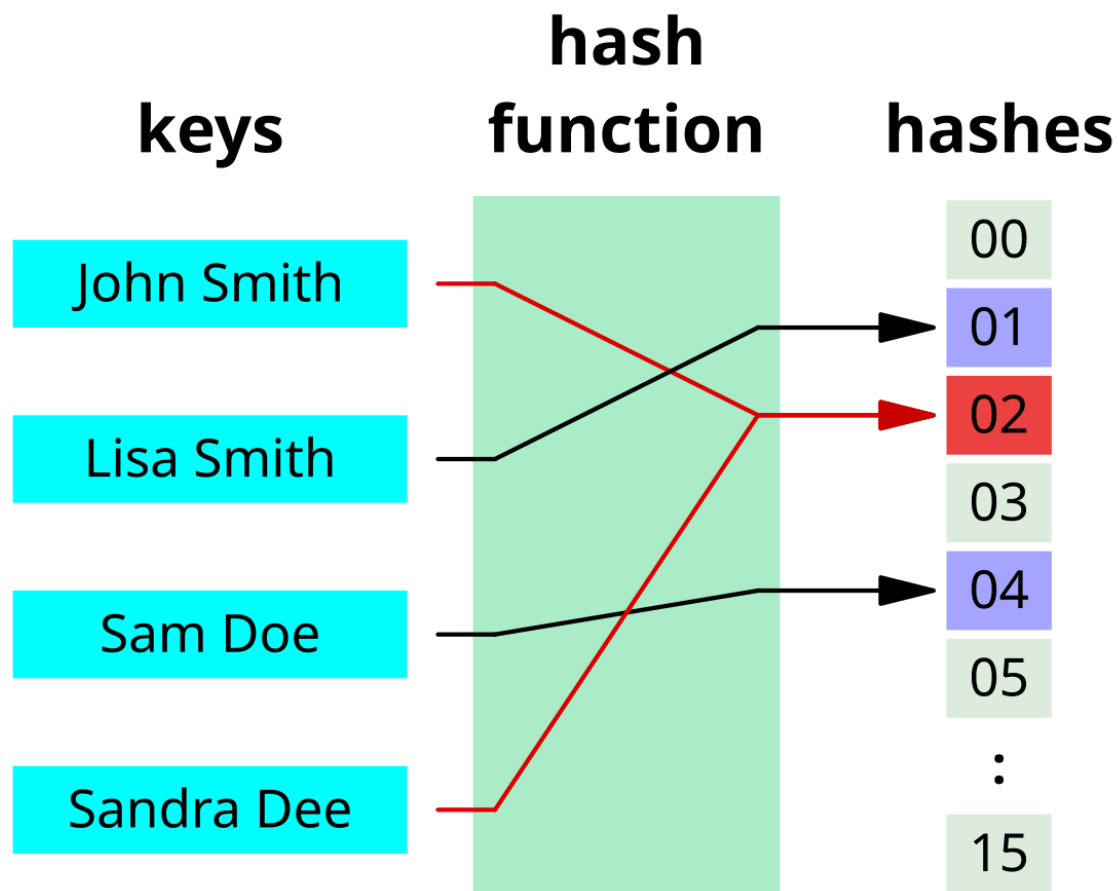
point1 * 4;
4 * point1;

```

Рекомендуется переопределять операторы только тогда когда это имеет понятный математический смысл. Никто не запрещает, например, реализовать сравнение значений через `-`, или запись текста через сложение файла и строки, или выводить сообщение в консоль через `*`, но это будет совершенно интуитивно не понятно и запутает код, поэтому не нужно так делать.

Хэш-функции

Хэш-функция — это функция, преобразующая массив входных данных произвольного размера в выходную битовую строку установленного размера в соответствии с определённым алгоритмом. Для того чтобы стало понятнее, давайте рассмотрим пример хэш-функции на рисунке. Эта хэш-функции преобразует строку произвольного размера в хэш фиксированного размера, который может принимать 16 значений от 0 до 15 (то есть его длина 4 бита). В идеале, разные строки должны иметь разный хэш, но в реальности количество всех возможных строк намного больше чем 16, поэтому неизбежно будут происходить ситуации, когда разные строки будут иметь одинаковый хэш (John Smith и Sandra Dee), такие ситуации называются **коллизиями**.



Давайте попробуем реализовать свою простейшую хэш-функцию. Будем просто складывать коды символов, из которых состоит строка, а потом возьмем остаток от деления на 256. Так мы получим хэш фиксированной длины в диапазоне от 0 до 255 (1 байт).

```
int Hash(string text)
{
    var hash = 0;
    foreach(var c in text)
        hash += (int)c;
    return hash % 256;
}
```

Эта функция будет переводить строку любого размера в число фиксированного размера, однако, такая функция будет давать много коллизий. Например, если мы поменяем порядок символов в строке, то хэш получится такой же. Поэтому обычно использую более качественные хэш функции, которые минимизируют количество коллизий, такую как **SHA-256** или уже устаревшие **MD5** и **SHA-1**.

Хэши имеют широкое применение. Например, у нас есть файловое хранилище и мы хотим загружать на него только уникальные файлы, чтобы избежать дублирования и сэкономить место. Для этого нам придется каждый раз сравнивать загружаемый файл

со всеми файлами, которые есть в хранилище, что может потребовать очень много вычислительных ресурсов и времени. Но можно поступить по другому. Вместо того, чтобы сравнивать файлы, можно рассчитать хэш для каждого файла и при загрузке файла сравнивать его хэш с хэшами остальных файлов. Сравнить небольшие хэши намного проще и быстрее, чем сравнивать большие файлы целиком. Если хэши у файлов отличаются, то можно сразу уверенно сказать, что они разные, если хэши одинаковые, то тогда файлы нужно сравнить полностью, на тот случай, если произошла коллизия. Таким же образом можно отслеживать изменения в файлах. Если внести даже самые маленькие изменения в файл, его хэш изменится очень сильно.

Еще одно применение хэшей связано с паролями. Когда вы регистрируетесь на каком-то сервисе, ваши данные, в том числе и пароль, сохраняются в базе данных. Но хранить пароли в открытом виде опасно, т.к. в случае утечки данных, злоумышленники смогут получить доступ ко всем учетным записям пользователей. Для того, чтобы этого избежать, в базе данных хранят хэши паролей, а не сами пароли. Когда вы вводите пароль, он хэшируется и сравнивается с хэшем из базы данных, если хэши одинаковые, то вы получаете доступ. Так же для большей безопасности в хэш-функции используют **соль** — специальное число, которое добавляется к паролю. Если база данных попадет в руки злоумышленников, они не смогут взломать учетные записи пользователей, т.к. им придется делать полный перебор всех возможных паролей, чтобы найти тот, который дает такой же хэш.

Словарь

Еще одним применением хэшей является такая структура данных как **хэш-таблица**. Предположим мы хотим реализовать ассоциативный массив, т.е. массив состоящий из пар *ключ-значение*, по сути это просто словарь. Как это можно сделать? Самый простой способ — просто хранить пары ключ-значение в списке. Давайте реализуем это:

```
//K - тип ключа
//V - тип значения
class MyDictionary<K, V>
{
    //список пар ключ-значение
    List<(K, V)> collection = new List<(K, V)>();

    public V this[K key]
    {
        get
        {
            foreach (var item in collection)
            {
                if (item.Item1.Equals(key))
                    return item.Item2;
            }
        }
    }
}
```

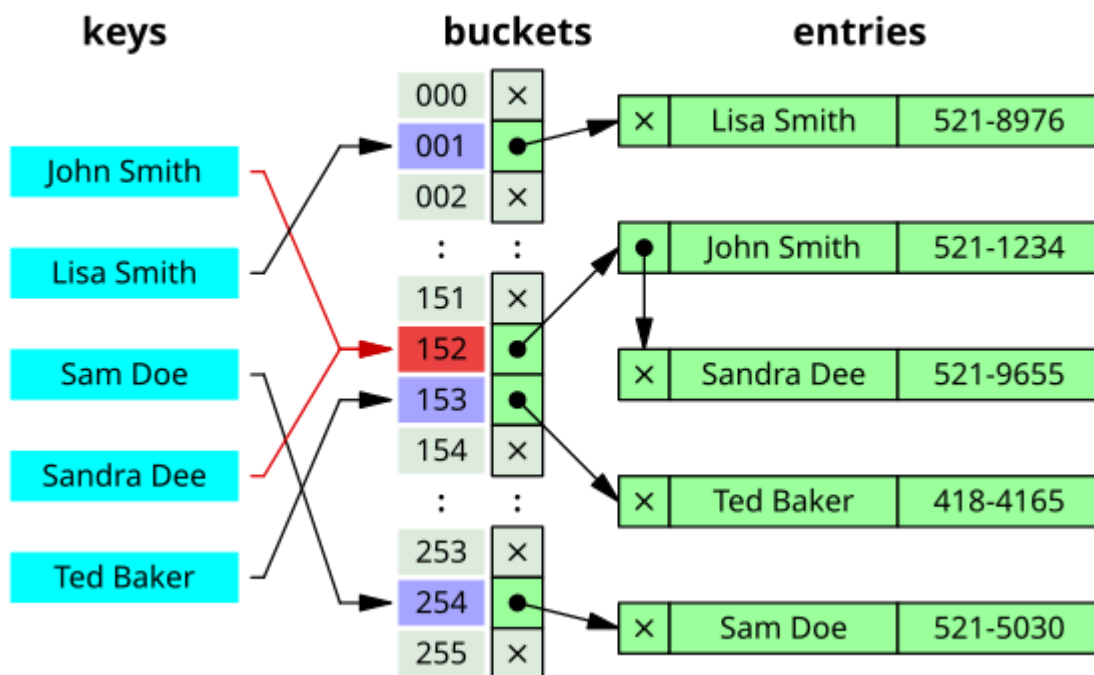
```

        throw new KeyNotFoundException();
    }
    set
    {
        bool valueFound = false;
        //ищем существующую запись
        for (int i = 0; i < collection.Count; i++)
        {
            if (collection[i].Item1.Equals(key))
            {
                collection[i] = (key, value);
                valueFound = true;
                break;
            }
        }
        //если не находим - создаем новую
        if (!valueFound)
        {
            collection.Add((key, value));
        }
    }
}

```

Такой способ будет работать, но он не самым эффективным, потому что приходится перебирать в цикле все записи для поиска нужной, поэтому при увеличении размера словаря, сложность чтения и записи будет возрастать линейно $O(n)$. С помощью хэш-таблиц можно реализовать ассоциативный массив, который будет работать намного эффективнее. Именно на этой структуре данных основан класс `Dictionary` в языке C#. Суть хэш-таблицы состоит в том, что мы хэшируем значение ключа и связываем этот хэш с парой ключ-значение. В чем преимущества данного подхода? Вместо того, чтобы искать нужную запись полным перебором всех ключей, мы можем просто получить ее по хэшу, это тоже самое, что получить значение в массиве по индексу. Сложность этой операции не зависит от размера массива, т.е. является константой $O(1)$.

Но что делать в случае коллизии, когда несколько ключей имеют одинаковый хэш? В этом случае хэш указывает не на одну запись, а на связный список записей с одинаковым хэшем. Поиск нужной записи происходит в два этапа, сначала находим по хэшу список записей, потом делаем полный перебор этих записей в поисках нужной. В худшем случае, когда все записи в хэш-таблице имеют одинаковый хэш, сложность поиска может стать линейной $O(n)$, но это очень маловероятное событие.



Ключами словаря могут быть не только строки, а любой тип данных. Как для них рассчитывается хэш? Если мы вернемся к классу `object`, то увидим в нем виртуальный метод `GetHashCode`.

```
namespace System
{
    ...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()...
    }
}
```

`GetHashCode` связан с `Equals`, который мы уже рассматривали. Если метод `Equals` указывает на то, что объекты одинаковые, то и `GetHashCode` должен давать для них одинаковый хэш. По умолчанию классы хэшируются по ссылке, а структуры по значениям полей. Давайте вернемся к примеру с классом точки, у которой мы переопределили метод `Equals` и также изменим `GetHashCode`, для того чтобы точки с одинаковыми координатами давали одинаковый хэш:

```

class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public override bool Equals(object obj)
    {
        var point = obj as Point;
        return X == point.X && Y == point.Y;
    }

    public override int GetHashCode()
    {
        //хэш будет рассчитываться из координат точки
        return GetHashCode.Combine(X, Y);
    }
}

```

Теперь мы можем писать так:

```

var point1 = new Point { X = 1, Y = 1 };
var dictionary = new Dictionary<Point, string>();
dictionary[point1] = "Test";
Console.WriteLine(dictionary[point1]); //Test
var point2 = new Point { X = 1, Y = 1 };
Console.WriteLine(dictionary[point2]); //Test

```