C#. Урок 10. Коллекции

Коллекции являются одним из наиболее часто используемых инструментов в разработке программного обеспечения. В этом уроке мы познакомимся с пространством имен *System.Collections.Generic,* коллекциями *List, Dictionary* и типом *Tuple.*

Коллекции
Самым примитивным способом хранения объектов в *C#* является использование массивов. Одной из основных проблем, с которой столкнется разработчик следуя такому подходу, является то, что массивы не предоставляют инструментов для динамического изменения размера. В языке *C#* есть два пространства имен для работы со структурами данных:

- *System.Collections;*
- *System.Collections.Generic.*

Первое из них – *System.Collections* предоставляет структуры данных для хранения объектов типа *Object*. У этого решения есть две основных проблемы – это производительность и безопасность типов. В настоящее время не рекомендуется использовать объекты классов из *System.Collections*.
Для решения указанных выше проблем *Microsoft* были разработаны коллекции с обобщенными типами (их ещё называют дженерики), они расположены в пространстве имен *System.Collections.Generic*. Суть их заключается в том, что вы не просто создает объект класса *List,* но и указываете, объекты какого типа будут в нем храниться, делается это так: *List<T>*, где *T* может быть *int, string, double* или какой-то ваш собственный класс.

Коллекции в языке *C#*. Пространство имен *System.Collections.Generic*
Пространство *System.Collections.Generic* содержит большой набор коллекций, которые позволяют удобно и эффективно решать широкий круг задач. Ниже, в таблице, перечислены некоторые из обобщенных классов с указанием интерфейсов, которые они реализуют.

Обобщенный класс	**Основные интерфейсы**	**Описание**
List<T>	*ICollection<T>, IEnumerable<T>, IList<T>*	Список элементов с динамически изменяемым размером
Dictionary<TKey, TValue>	*ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>*	Коллекция элементов связанных через уникальный ключ
Queue<T>	*ICollection, IEnumerable<T>*	Очередь – список,

		работающий по алгоритму *FIFO*
Stack<T>	*ICollection, IEnumerable<T>*	Стэк – список, работающий по алгоритму *LIFO*
SortedList<TKey,TValue>	*IComparer<T>*, *ICollection<KeyValuePair<TKey,TValue>>*, *IDictionary<TKey,TValue>*	Коллекция пар “ключ-значение”, упорядоченных по ключу

Познакомимся поближе с несколькими классами из приведенной таблицы.

Класс *List<T>*

Начнем наше знакомство с коллекциями с класса *List<T>*. Эта коллекция является аналогом типизированного массива, который может динамически расширяться. В качестве типа можно указать любой встроенный либо пользовательский тип.

Создание объекта класса *List<T>*

Можно создать пустой список и добавить в него элементы позже, с помощью метода *Add()*:

```
List<int> numsList = new List<int>();
numsList.Add(1);
```

Либо воспользоваться синтаксисом, позволяющем указать набор объектов, который будет храниться в списке:

```
List<int> nums = new List<int> {1, 2, 3, 4, 5};
var words = new List<string> {"one", "two", "three"};
```

Работа с объектами *List<T>*

Ниже приведены таблицы, в которых перечислены некоторые полезные свойства и методы класса *List<T>*.

Свойства класса *List<T>*

Свойство	**Описание**
Count	Количество элементов в списке
Capacity	Емкость списка – количество элементов, которое может вместить список без изменения размера

```
Console.WriteLine("Свойства");
Console.WriteLine($"- Count: nums.Count = {nums.Count}");
Console.WriteLine($"- Capacity: nums.Capacity = {nums.Capacity}");
```

Методы класса *List<T>*

Метод	**Описание**
Add(T)	Добавляет элемент к списку
BinarySearch(T)	Выполняет поиск по списку
Clear()	Очистка списка
Contains(T)	Возвращает *true*, если список содержит указанный элемент
IndexOf(T)	Возвращает индекс переданного элемента
ForEach(Action<T>)	Выполняет указанное действие для всех элементов списка
Insert(Int32, T)	Вставляет элемент в указанную позицию
Find(Predicate<T>)	Осуществляет поиск первого элемент, для которого выполняется заданный предикат
Remove(T)	Удаляет указанный элемент из списка
RemoveAt(Int32)	Удаляет элемент из заданной позиции
Sort()	Сортирует список
Reverse()	Меняет порядок расположения элементов на противоположный

```
Console.WriteLine($"nums: {ListToString(nums)}");
nums.Add(6);
Console.WriteLine($"nums.Add(6): {ListToString(nums)}");
Console.WriteLine($"words.BinarySearch(\"two\"): {words.BinarySearch("two")}");
Console.WriteLine($"nums.Contains(10): {nums.Contains(10)}");
Console.WriteLine($"words.IndexOf(\"three\"): {words.IndexOf("three")}");
Console.WriteLine($"nums.ForEach(v => v * 10)");
nums.ForEach(v => Console.Write($"{v} => "));
nums.Insert(3, 7);
Console.WriteLine($"nums.Insert(3, 7): {ListToString(nums)}");
Console.WriteLine($"words.Find(v => v.Length == 3): {words.Find(v => v.Length == 3)}");
words.Remove("two");
Console.WriteLine($"words.Remove(\"two\"): {ListToString(words)}");
```

Код метода *ListToString*:

```
static private string ListToString<T>(List<T> list) =>
"{" + string.Join(", ", list.ToArray()) + "}";
```

Далее приведен пример работы со списком, в котором хранятся объекты пользовательского типа. Создадим класс *Player*, имеющий свойства: *Name* и *Skill*.

```
class Player
{
public string Name { get; set; }
public string Skill { get; set; }
}
```

Создадим список игроков и выполним с ним ряд действий:

```
Console.WriteLine("Работа с пользовательским типом");
List<Player> players = new List<Player> {
new Player { Name = "Psy", Skill = "Monster"},
new Player { Name = "Kubik", Skill = "Soldier"},
new Player { Name = "Triver", Skill = "Middle"},
new Player { Name = "Terminator", Skill = "Very High"}
};
Console.WriteLine("Количество элементов в players:{0}", players.Count);
//Добавим новый элемент списка players
players.Insert(1, new Player { Name = "Butterfly", Skill = "flutter like a butterfly, pity like a bee"});
//Посмотрим на все элементы списка
players.ForEach(p => Console.WriteLine($"{p.Name}, skill: {p.Skill}"));
```

Класс *Dictionary<TKey,TValue>*

Класс *Dictionary* реализует структуру данных **Отображение**, которую иногда называют Словарь или Ассоциативный массив. Идея довольно проста: в обычном массиве доступ к данным мы получаем через целочисленный индекс, в словаре используется ключ, который может быть числом, строкой или любым другим типом данных, который реализует метод *GetHashCode()*. При добавлении нового объекта в такую коллекцию для него указывается уникальный ключ, который используется для последующего доступа к нему.

Создание объекта класса *Dictionary*

Пустой словарь:

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

Словарь с набором элементов:

```
var prodPrice = new Dictionary<string, double>()
{
["bread"] = 23.3,
["apple"] = 45.2
};
Console.WriteLine($"bread price: {prodPrice["bread"]}");
```

Работа с объектами *Dictionary*

Рассмотрим некоторые из свойств и методов класса *Dictionary<TKey, TValue>*. Полное описание возможностей этого класса вы можете найти на официальной странице Microsoft.

Свойства класса *Dictionary*

Свойство	**Описание**
Count	Количество объектов в словаре
Keys	Ключи словаря
Values	Значения элементов словаря

```
Console.WriteLine("Свойства");
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
Console.WriteLine($"Count: {prodPrice.Count}");
Console.WriteLine($"Keys: {ListToString(prodPrice.Keys.ToList<string>())}");
Console.WriteLine($"Values: {ListToString(prodPrice.Values.ToList<double>())}");
```

Методы класса *Dictionary*

Метод	**Описание**
Add(TKey, TValue)	Добавляет в словарь элемент с заданным ключом и значением
Clear()	Удаляет из словаря все ключи и значения
ContainsValue(TValue)	Проверяет наличие в словаре указанного значения
ContainsKey(TKey)	Проверяет наличие в словаре указанного ключа
GetEnumerator()	Возвращает перечислитель для перебора элементов словаря
Remove(TKey)	Удаляет элемент с указанным ключом
TryAdd(TKey, TValue)	Метод, реализующий попытку добавить в словарь элемент с заданным ключом и значением
TryGetValue(TKey, TValue)	Метод, реализующий попытку получить значение по заданному ключу

```
prodPrice.Add("tomate", 11.2);
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
var isExistValue = prodPrice.ContainsValue(11.2);
Console.WriteLine($"isExistValue = {isExistValue}");
var isExistKey = prodPrice.ContainsKey("tomate");
Console.WriteLine($"isExistKey = {isExistKey}");
```

```
prodPrice.Remove("bread");
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
var isOrangeAdded = prodPrice.TryAdd("orange", 20.1);
Console.WriteLine($"isOrangeAdded = {isOrangeAdded}");
double orangePrice;
var isPriceGetted = prodPrice.TryGetValue("orange", out orangePrice);
Console.WriteLine($"isPriceGetted = {isPriceGetted}");
Console.WriteLine($"orangePrice = {orangePrice}");
prodPrice.Clear();
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
```

Кортежи *Tuple* и *ValueTuple*

Относительно недавним нововведением в языке *C#* (начиная с *C#* 7) являются кортежи. Кортежем называют структуру данных типа *Tuple* или *ValueTuple* (чуть ниже мы рассмотрим различия между ними), которые позволяют группировать объекты разных типов друг с другом. С практической точки зрения они являются удобным способом возврата из метода нескольких значений – это наиболее частый вариант использования кортежей.

Различия между *Tuple* и *ValueTuple* приведены в таблице ниже.

Tuple	**ValueTuple**
Ссылочный тип	Тип значение
Неизменяемый тип	Изменяемый тип
Элементы данных – это свойства	Элементы данных – это поля

Создание кортежей

Рассмотрим несколько вариантов создания кортежей.

Создание кортежа без явного и с явным указанием имен полей:

```
(string, int) p1 = ("John", 21);
(string Name, int Age) p2 = ("Mary", 23);
```

При этом для доступа к элементам кортежа в первом варианте используются свойства *Item* с числом, указывающем на порядок элемента, во втором – заданные пользователем имена:

```
Console.WriteLine($"p1: Name: {p1.Item1}, Age: {p1.Item2}");
Console.WriteLine($"p1: Name: {p2.Name}, Age: {p2.Age}");
```

Возможны следующие способы создания кортежей с явным заданием имен:

```
var p3 = (Name: "Alex", Age: 24);
var Name = "Lynda";
var Age = 25;
var p4 = (Name, Age);
Console.WriteLine($"p3: Name: {p3.Name}, Age: {p3.Age}");
Console.WriteLine($"p4: Name: {p4.Name}, Age: {p4.Age}");
```

При этом возможность обращаться через свойства *Item1* и *Item2* для созданных выше переменных остается:

```
Console.WriteLine($"p3: Name: {p3.Item1}, Age: {p3.Item2}");
Console.WriteLine($"p4: Name: {p4.Item1}, Age: {p4.Item2}");
```

Работа с кортежами

Как было сказано в начале раздела, кортежи можно возвращать в качестве результата работы метода. Пример метода, который сравнивает длину переданной строки с некоторым порогом и возвращает соответствующее *bool*-значение и целое число – длину строки:

```
static (bool isLonger, int count) LongerThenLimit(string str, int limit) =>
str.Length > limit ? (true, str.Length) : (false, str.Length);
```

Кортежи можно присваивать друг другу, при этом необходимо, чтобы соблюдались следующие условия:

- количество элементов в обоих кортежах одинаковое;
- типы соответствующих элементов совпадают, либо могут быть приведены друг к другу.

```
var p5 = ("Jane", 26);
(string, int) p6 = p5;
Console.WriteLine($"p6: Name: {p6.Item1}, Age: {p6.Item2}");
```

Операцию присваивания можно использовать для деструкции кортежа.

```
(string name, int age) = p5;
Console.WriteLine($"Name: {name}, Age: {age}");
```