

МДК 01.02 ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ

**ТЕМА 2.2. РАЗРАБОТКА КОДА ПРОГРАММНОГО
ПРОДУКТА НА УРОВНЕ МОДУЛЯ.**

ТЕМА:

**Обобщения (generics).
Коллекции.**

Ломова Л.А.

Терминология

В **C#** существует два механизма повторного использования кода в разных типах: наследование и обобщения (**generics**).

- **Наследование** делает возможным повторное использование благодаря применению базового класса, а обобщения — благодаря использованию шаблонов.
- **Шаблоны** - (**Generics C#** - обобщения) - это универсальный механизм позволяющий выполнять обработку данных принимая в качестве параметра тип этих данных.



Пространства имен для работы со структурами данных

- ***System.Collections;*** - предоставляет структуры данных для хранения объектов типа *Object* (проблемы – производительность и безопасность типов).
- ***System.Collections.Generic;*** - содержит коллекции с обобщенными типами.

Обобщения упрощают весь процесс, поскольку исключают необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных (расширяют возможности повторного использования кода).

Примеры обобщенных классов из *System.Collections.Generic*

Обобщенный класс	Описание
List<T>	Список элементов с динамически изменяемым размером
Dictionary<TKey, TValue>	Коллекция элементов связанных через уникальный ключ
Queue<T>	Очередь – список, работающий по алгоритму FIFO
Stack<T>	Стэк – список, работающий по алгоритму LIFO
SortedList<TKey, TValue>	Коллекция пар “ключ-значение”, упорядоченных по ключу

Класс *List*<T>

- Эта коллекция является аналогом типизированного массива (представляет простейший список однотипных объектов), который может динамически расширяться. В качестве типа можно указать любой встроенный, либо пользовательский тип.

Синтаксис:

List <T> имя;

List <T> имя = **new** List <T>(); // создание пустого списка List <T>

Примеры работы с List

1

```
List<int> nums = new List<int>(); // инициализация нового пустого списка List<T>
nums.Add(1); // добавление нового элемента
```


2

```
// инициализация списков List<>
List<int> nums = new List<int>() {1,2,3,4,5};
List<string> words = new List<string> { "one","two","three"};

nums.Add(32); // добавление нового элемента в конец списка
```

```
// вывод значений списка
Console.WriteLine("Содержимое List: ");
foreach (int item in nums)
{
    Console.WriteLine(item + " ");
}
```

```
Console.WriteLine("\nКоличество элементов List: " + nums.Count); // нумерация с нуля
```

 D:\temp\ConsoleApp1\bin\Debug\ConsoleApp1.exe

```
Содержимое List: 1 2 3 4 5 32
Количество элементов List: 6
```

Свойства класса *List<T>*:

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

Пример применения:

```
Console.WriteLine($"Элемент 32 - {nums.Contains(32).ToString()}"); // определение наличия в List элемента "32" (true)
```

Примеры методов класса *List<T>*:

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

Объединение, пересечение и разность коллекций

- метод **Except** – разность двух последовательностей;
- метод **Intersect** – пересечение двух последовательностей;
- метод **Union** – объединение последовательностей без повторений;
- метод **Concat** – объединение последовательностей с повторениями;
- метод **Distinct** – удаление дубликатов в наборе.

Множества

Множество – это совокупность объектов, рассматриваемая как одно целое.

Для работы с множествами в библиотеке классов .NET Framework имеются обобщённые классы – **HashSet<T>** и **SortedSet<T>**, которые находятся в пространстве имён System.Collections.Generic.

Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены.

Объявление:

```
HashSet<тип> a = new HashSet<тип>();  
SortedSet<тип> b = new SortedSet<тип>();
```

Множества

Set (множество) поддерживает базовые операции:

- Добавить элемент.
- Удалить элемент.
- Проверить, существует ли элемент.
- Перебрать все элементы.

Множества

`HashSet<T>` коллекция не сортируется и не может содержать дублирующиеся элементы.

`HashSet<T>` позволяет быстро определить, есть такой элемент или нет (быстро потому что, использует индекс, который вычисляется из хэш-кода элемента).

`HashSet<T>` класс может рассматриваться как `Dictionary<TKey,TValue>` Коллекция без значений.

`HashSet<T>` - имеет методы **Add**, **Remove**, **Contains**, но поскольку он использует хэш-реализацию, эти операции занимают 1 действие (методы **Contains** и **Remove** в `List<T>` занимает n-действий).

`HashSet<T>` имеет методы:

- `UnionWith` (**объединение** элементов с другим `HashSet<T>`);
- `IntersectWith` (**пересечение** элементов с другим `HashSet<T>`);
- `ExceptWith` (**разность** элементов с другим `HashSet<T>`).

Объединение, пересечение и разность коллекций. Примеры

```
List<int> key = new List<int>() {1, 2, 3, 4, 5};  
List<int> val = new List<int>() { 1, 22, 33, 44, 5 };  
  
var v1 = key.Except(val); //разность двух списков, т.е. все неповторяющиеся значения из key  
var v2 = key.Intersect(val); // пересечение двух списков  
var v3 = key.Union(val); // объединение двух списков без повторений  
var v4 = key.Concat(val); // простое объединение списков (с повторениями)  
var v5 = key.Concat(val).Distinct(); // удаление дубликатов
```

Как работает метод Zip?

Коллекция Dictionary<T, V>

- Класс *Dictionary* реализует структуру данных **Отображение**, которую иногда называют Словарь или Ассоциативный массив. Словарь хранит объекты, которые представляют пару ключ-значение.

Можно также свободно добавлять и удалять элементы, подобно тому, как это делается в *List<T>*, но без расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

Словари имеют динамический характер, расширяясь по мере необходимости.

Конструкторы класса Dictionary<TKey, TValue>

- ▶ **public Dictionary();** // создается пустой словарь с выбираемой по умолчанию первоначальной емкостью
- ▶ **public Dictionary(IDictionary<TKey, TValue> dictionary);** // создается словарь с указанным количеством элементов dictionary
- ▶ **public Dictionary(int capacity);** // 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

Примеры Dictionary<TKey, TValue>

// пример 1

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

// пример 2

```
var prodPrice = new Dictionary<string, double>() // создание пустого словаря
```

```
{
```

```
    ["bread"] = 45.99, // задание пары "ключ - значение"
```

```
    ["milk"] = 56.0
```

```
};
```

```
Console.WriteLine($"Цена хлеба: {prodPrice["bread"]} рублей. ");
```

// пример 3

```
Dictionary<int, string> countries = new Dictionary<int, string>(3);
```

```
countries.Add(1, "Russia");
```

```
countries.Add(2, "China");
```

```
countries.Add(3, "USA");
```

// пример 4

```
Dictionary<int, string> countries = new Dictionary<int, string>
```

```
{
```

```
    [1] = "Russia",
```

```
    [2] = "China",
```

```
    [3] = "USA"
```

```
};
```


Свойства класса *Dictionary*

Свойство	Описание
<i>Count</i>	Количество объектов в словаре
<i>Keys</i>	Ключи словаря
<i>Values</i>	Значения элементов словаря

Примеры методов класса *Dictionary*:

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

Примеры Dictionary<TKey, TValue>

Обращение к ключу – «**Key**». Пример: counties.Key

Обращение к значению – «**Value**». Пример: counties.Value

Определяет пару «ключ-значение», которая м.б. задана или получена

```
foreach(KeyValuePair<int,string> x in countries) // перебор всех значений Dictionary
{
    Console.WriteLine($"{x.Key} - {x.Value}");
}
```

ИЛИ

```
foreach(var x in countries) // перебор всех значений Dictionary
{
    Console.WriteLine($"{x.Key} - {x.Value}");
}
```

```
// получение элемента по ключу
string country = countries[3];
// изменение объекта
countries[3] = "Spain"; // если его нет - добавление, если он есть - изменение
// удаление по ключу
countries.Remove(2);
```

Самостоятельно:

1. Создать:

- список товаров для покупки.
- словарь: Семейство (фамилия имя, возраст).

2. Внести изменения (добавить новый элемент; изменить уже существующий), найти значение.

3. Вывести содержимое на экран.

4. Соединить два списка в один (Union, Concat, Zip), вывести на экран результат.

5. Создать словарь с 5 элементами. Поменять местами первый и последний элемент объекта. Удалить второй элемент. Добавить в конец ключ «new_key» со значением «new_value». Вывести итоговый словарь.