Тема 10. Коллекции.

Учебные вопросы:

1. Введение в коллекции.
2. Списки.
3. Словари.
4. Множества.

1. Введение в коллекции.

Коллекции в программировании — это структуры данных, которые позволяют хранить и управлять множеством объектов как единым целым. В C# коллекции предоставляют способ работы с группами объектов, независимо от их типа, и обеспечивают удобный доступ, модификацию и поиск элементов.

Коллекция — это упорядоченное или неупорядоченное собрание объектов одного или различных типов, которые могут быть обработаны как единая сущность. Коллекции могут включать в себя простые типы (например, целые числа или строки) или сложные объекты.

Коллекции играют важную роль в программировании по следующим причинам:

- **Упрощение работы с данными**: Коллекции позволяют легко хранить, организовывать и манипулировать множеством элементов без необходимости создавать отдельные переменные для каждого элемента.
- **Динамическое управление данными**: В отличие от массивов, которые имеют фиксированный размер, большинство коллекций в C# могут динамически изменять размер в процессе выполнения программы, добавляя или удаляя элементы по мере необходимости.
- **Гибкость и функциональность**: Коллекции предоставляют встроенные методы и свойства для выполнения различных операций, таких как добавление, удаление, сортировка, фильтрация и поиск элементов, что упрощает и ускоряет разработку.
- **Мощность и производительность**: Некоторые коллекции оптимизированы для конкретных задач.

Обзор типов коллекций в C#

В C# существует несколько типов коллекций, каждая из которых предназначена для определенных задач:

- **Массивы** (Array). Фиксированного размера, предназначены для хранения данных одного типа. Размер массива задается при его создании и не может быть изменен.
- **Списки** (List<T>). Представляют собой динамические массивы, которые автоматически изменяют размер при добавлении или удалении элементов. Они обеспечивают быстрый доступ к элементам по индексу.

- **Словари** (Dictionary<TKey, TValue>). Хранят пары "ключ-значение" и обеспечивают быстрый доступ к элементам по ключу. Ключи должны быть уникальными.
- **Множества** (HashSet<T>). Хранят уникальные элементы, не допускают дубликатов и обеспечивают быструю проверку на наличие элемента в коллекции.
- **Очереди** (Queue<T>). Реализуют принцип FIFO (First In, First Out), где первый добавленный элемент будет первым извлечен.
- **Стэки** (Stack<T>). Реализуют принцип LIFO (Last In, First Out), где последний добавленный элемент будет первым извлечен.

- **Связанные списки** (LinkedList<T>). Состоят из элементов (узлов), каждый из которых содержит ссылку на следующий и предыдущий элементы. Подходит для частых операций добавления и удаления в середине коллекции.
- **ObservableCollection**<T>. Специальный тип коллекции, который уведомляет о изменениях (добавление, удаление, изменение элементов). Часто используется в приложениях с графическим интерфейсом.
- **Кортежи (tuples)** - это неизменяемые коллекции, которые могут хранить значения разных типов данных.

Каждый тип коллекции имеет свои особенности, и выбор подходящей коллекции зависит от конкретной задачи, которую нужно решить в программировании.

Основная часть коллекций находится в пространстве имен **System.Collections.Generic**

2. Списки.

Список (List<T>**)**— это обобщенный класс в C#, представляющий динамически изменяемый список объектов определенного типа.

В отличие от массивов, которые имеют фиксированный размер, списки могут автоматически изменять свой размер при добавлении или удалении элементов.

Основные характеристики списков:

- **Типобезопасность**: List<T> является обобщенным, что означает, что он может хранить элементы только определенного типа . Это позволяет избежать ошибок времени выполнения, связанных с неверными типами данных.
- **Динамический размер**: List<T> автоматически изменяет свой размер по мере необходимости, когда элементы добавляются или удаляются. Внутренне List<T> использует массив, который расширяется при превышении его емкости.
- **Доступ по индексу**: Списки поддерживают доступ к элементам по индексу, что делает их похожими на массивы. Индексация начинается с нуля.
- **Вставка, удаление** и поиск элементов: List<T> предоставляет различные методы для работы с элементами, включая вставку, удаление и поиск.

Создание и инициализация списков.

Для создания списка необходимо указать тип элементов, которые он будет хранить.

Создать список можно несколькими способами:

```
// Пустой список целых чисел
List<int> numbers = new List<int>();

// Список строк с начальной инициализацией
List<string> fruits = new List<string> { "Apple", "Banana", "Cherry" };
```

Основные методы списков:

Добавление элементов.

- **Add():** Добавляет элемент в конец списка.
- **AddRange():** Добавляет элементы из коллекции в конец списка.

```
numbers.Add(10);
numbers.AddRange(new int[] { 20, 30, 40 });
```

Вставка элементов.

- **Insert():** вставка элемента в список на определенную позицию

```
// Создаем список строк
List<string> fruits = new List<string> { "Apple", "Banana", "Cherry" };


// Вставляем элемент на позицию 1 (между "Apple" и "Banana")
fruits.Insert(1, "Orange");
```

Удаление элементов

- **Remove():** Удаляет первое вхождение элемента из списка.
- **RemoveAt():** Удаляет элемент по указанному индексу.
- **Clear():** Удаляет все элементы из списка

```
fruits.Remove("Banana");
fruits.RemoveAt(0);
fruits.Clear();
```

Поиск элементов

- **Contains():** Проверяет, содержится ли элемент в списке.
- **IndexOf():** Возвращает индекс первого вхождения элемента или -1, если элемент не найден.
- **Find():** Возвращает первый элемент, удовлетворяющий условию, заданному предикатом.

```
bool hasApple = fruits.Contains("Apple");
int index = fruits.IndexOf("Cherry");
string fruit = fruits.Find(f => f.StartsWith("B"));
```

Другие полезные методы и свойства

- **Count**: Возвращает количество элементов в списке.
- **Sort()**: Сортирует элементы списка.
- **Reverse()**: Меняет порядок элементов в списке на обратный.
- **ToArray()**: Преобразует список в массив.

```
int count = numbers.Count;
fruits.Sort();
numbers.Reverse();
int[] array = numbers.ToArray();
```

Когда использовать списки?

- Когда заранее не известно количество элементов.
- Когда требуется часто добавлять или удалять элементы.
- Когда необходимо хранить элементы в определенном порядке.
- Когда нужна возможность быстрого доступа к элементам по индексу.

3. Словари.

Словарь (Dictionary) в C# - это коллекция, которая хранит пары "ключ-значение". Каждый элемент в словаре уникально идентифицируется своим ключом. Словари особенно полезны, когда нужно быстро найти значение по заданному ключу.

Ключи в словаре должны быть уникальными, а значения могут быть любого типа.

Основные характеристики словарей:

- **Типобезопасность**: Словарь является обобщенным, поэтому ключи и значения должны соответствовать указанным типам **TKey** и **TValue**, что позволяет избежать ошибок типов во время выполнения.
- **Уникальные ключи**: Каждый ключ в словаре должен быть уникальным. Если попытаться добавить элемент с уже существующим ключом, это вызовет исключение или перезапишет существующее значение, в зависимости от метода.
- **Быстрый доступ**: В отличие от списков, где элементы индексируются по порядку, в словаре доступ к значениям осуществляется по ключу. Это делает поиск и извлечение данных очень быстрыми.

Создание и инициализация словарей.

Для создания словаря необходимо указать типы ключей и значений:

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


// Словарь с начальной инициализацией
Dictionary<int, string> dict = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
    { 3, "Three" }
};
```

Добавление элементов

Для добавления пары "ключ-значение" используется метод **Add()**:

```
dict.Add(4, "Four");
```

Доступ к элементам

dict[key]: Позволяет получить или установить значение по ключу. Если ключ отсутствует, при попытке получения значения будет выброшено исключение KeyNotFoundException.

```
string value = dict[2]; // "Two"

dict[2] = "Deux"; // Изменяем значение по ключу
```

Удаление элементов

Remove(TKey key): Удаляет элемент с указанным ключом из словаря. Если ключ отсутствует, ничего не происходит.

```
dict.Remove(3);
```

Проверка наличия элементов

ContainsKey(TKey key): Возвращает **true**, если ключ существует в словаре.

ContainsValue(TValue value): Возвращает **true**, если значение существует в словаре.

```
bool hasKey = dict.ContainsKey(1); // true
bool hasValue = dict.ContainsValue("Four"); // true
```

Перебор элементов

foreach цикл позволяет перебрать все пары "ключ-значение" в словаре.

```
foreach (var kvp in dict)
{
    Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
}
```

Количество элементов

Count: Возвращает количество пар "ключ-значение" в словаре.

```
int count = dict.Count;
```

Когда использовать словари.

Словарь используется, когда необходимо быстро находить данные по ключу, а порядок элементов не имеет значения.

Это идеальный выбор для задач, где необходимо сопоставить уникальные ключи с определенными значениями, например, для хранения и поиска данных о пользователях, товарах, конфигурациях и т.д.

Словари - это мощный инструмент для хранения и поиска данных в C#. Они широко используются в различных приложениях, таких как:

- Кэширование данных.
- Реализация словарей и энциклопедий.
- Создание конфигурационных файлов.
- И многое другое.

4. Множества.

Множества в C# предоставляют коллекции, где элементы хранятся без дубликатов, и обеспечивают быстрые операции по добавлению, удалению и проверке наличия элементов.

Основные типы множеств

HashSet<T>:

- Основан на хеш-таблице.
- Обеспечивает быстрый доступ к элементам.
- Не гарантирует порядка элементов.

SortedSet<T>:

- Элементы автоматически сортируются при добавлении.
- Использует двоичное дерево поиска.
- Обеспечивает доступ к элементам в отсортированном порядке.

Основные операции над множествами:

- Добавление элемента: Add
- Удаление элемента: Remove
- Проверка на наличие элемента: Contains
- Объединение множеств: UnionWith
- Пересечение множеств: IntersectWith
- Разность множеств: ExceptWith

Создание множества:

```
HashSet<int> set1 = new HashSet<int>();
HashSet<int> set2 = new HashSet<int> { 1, 8, 7, 4, 8}; // 1,8,7,4
SortedSet<int> set3 = new SortedSet<int> { 6, 1, 9, 3, 6}; // 1,3,6,9
```

Добавление элементов:

```
hashSet.Add(5);  // Добавляет элемент 5
sortedSet.Add(0); // Добавляет элемент 0
```

Удаление элементов:

```
hashSet.Remove(2);  // Удаляет элемент 2
sortedSet.Remove(3); // Удаляет элемент 3
```

Проверка наличия элементов:

```
bool contains = hashSet.Contains(1);  // Проверяет, содержится ли элемент 1
```

Перебор элементов:

```
foreach (int item in hashSet)
{
    Console.WriteLine(item);
}

foreach (int item in sortedSet)
{
    Console.WriteLine(item); // Элементы будут выведены в отсортированном порядке
}
```

Операции над множествами:

- **Объединение** (Union): Возвращает множество, содержащее все элементы обоих множеств

```
hashSet.UnionWith(otherSet);
```

- **Пересечение** (Intersection): Возвращает множество, содержащее только общие элементы.

```
hashSet.IntersectWith(otherSet);
```

- **Разность** (Difference): Возвращает множество, содержащее элементы, которые есть в одном множестве, но отсутствуют в другом.

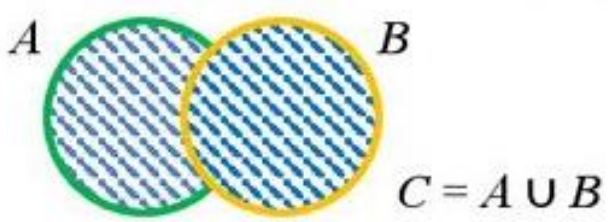
```
hashSet.ExceptWith(otherSet);
```

- **Симметрическая разность** (SymmetricDifference): Возвращает множество, содержащее элементы, которые присутствуют только в одном из двух множеств.

```
hashSet.SymmetricExceptWith(otherSet);
```

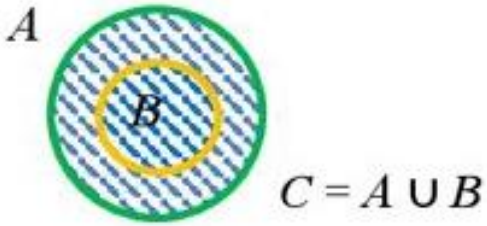
Объединение множеств

$A \cup B$ = {все элементы, принадлежащие хотя бы одному из множеств A и B}



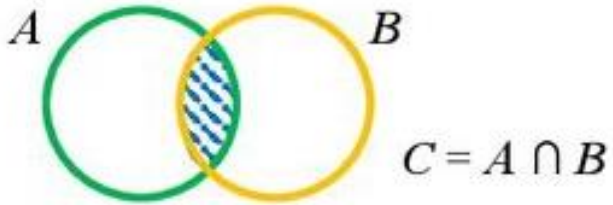
$C = A \cup B$

$C = A \cup B$

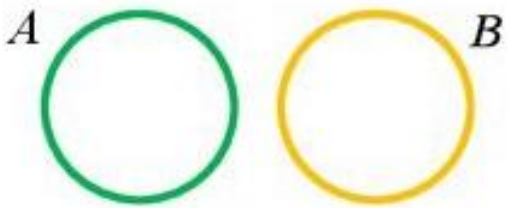


$C = A \cup B$

Пересечение множеств $A \cap B$ = {все элементы, принадлежащие как A, так и B}



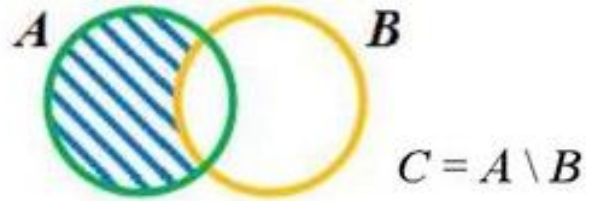
$C = A \cap B$



$C = \varnothing$

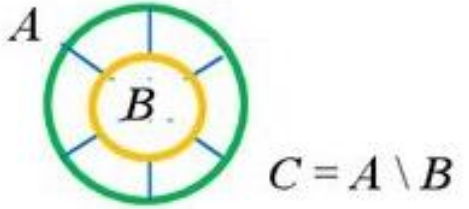
$C = A \cap B = B$

Разность множеств $A \setminus B = \{x: x \in A, x \notin B\}$



$C = A \setminus B$

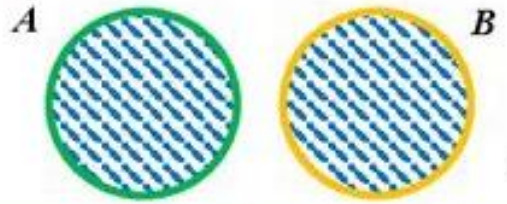
$C = A \setminus B = A$



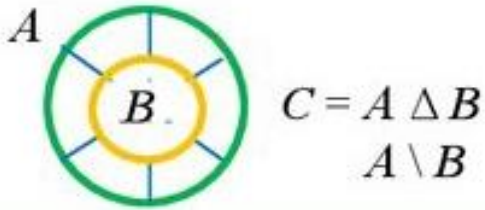
$C = A \setminus B$

Симметрическая разность множеств $A \Delta B = (A \setminus B) \cup (B \setminus A)$

$C = A \Delta B$



$C = A \Delta B = A \cup B$



$C = A \Delta B = A \setminus B$

Когда использовать множества

Множества полезны, когда вам нужно работать с уникальными элементами и быстро выполнять операции, такие как объединение, пересечение или разность. Например, множества часто используются в задачах, связанных с обработкой данных, когда важно исключить дубликаты или сравнить группы данных.

Когда использовать какой класс?

- HashSet<T>: Если порядок элементов не важен, а важна высокая скорость операций добавления, удаления и поиска.
- SortedSet<T>: Если необходимо поддерживать элементы в отсортированном порядке и часто выполнять операции поиска по диапазону.

Заключение.

Коллекции в C# предоставляют богатый набор инструментов для работы с данными.

Выбор конкретной коллекции зависит от задач, которые необходимо решить: будь то упорядоченное хранение данных, быстрый доступ по ключу или работа с уникальными элементами.

Правильное использование коллекций позволяет оптимизировать код, улучшить его читаемость и повысить производительность приложения.

Контрольные вопросы:

- Что такое коллекции в программировании и какие преимущества они предоставляют?
- В чем разница между массивами и списками в C#?
- Какой метод используется для добавления элемента в конец списка List<T>?
- В чем разница между массивами и словарями в C#?
- Какие основные методы предоставляются словарем Dictionary<TKey, TValue> для работы с элементами?
- Какой метод позволяет проверить наличие ключа в словаре?
- Чем множество отличается от списка?
- Чем отличаются множества HashSet<T> и SortedSet<T>?
- Какие операции можно выполнять над множествами в C#?
- В каких случаях лучше использовать множества вместо списков или словарей?

Домашнее задание:

1. Повторить материал лекции.
2. Задача 1: Сумма элементов списка. Создайте список с несколькими элементами. Пройдитесь по списку с использованием цикла for или foreach. Вычислите сумму всех элементов и выведите её на экран.
3. Задача 2: Создание словаря и доступ к элементам. Создайте словарь, где ключом будет строка (имя), а значением — целое число (возраст). Используйте ключ для получения значения (возраста) и выведите его на экран.

4. Задача 3: Добавление элементов в множество. Создайте пустое множество. Добавьте в него несколько уникальных элементов. Выведите все элементы множества.

Материалы лекций:

https://github.com/ShViktor72/Education2025