**Классы, коллекции, Классы коллекций ArrayList, Stack, Queue, Hashtable, SortedList**

Коллекции в C# — это структуры данных, предназначенные для хранения, управления и манипулирования наборами объектов. Они позволяют хранить и обрабатывать группы элементов, такие как объекты или значения, в единой структуре. В отличие от массивов, которые имеют фиксированный размер, коллекции могут динамически изменять свой размер по мере добавления или удаления элементов.

**Ключевые характеристики коллекций:**

Динамическое управление размером: коллекции автоматически расширяются или сжимаются в зависимости от количества элементов, что делает их более гибкими по сравнению с массивами.

Разнообразие типов данных: большинство коллекций работают с объектами типа object, что позволяет хранить элементы любого типа. Однако существуют обобщённые коллекции, которые обеспечивают строгое соответствие типу.

Стандартные методы: коллекции предоставляют встроенные методы для добавления, удаления, поиска и сортировки элементов.

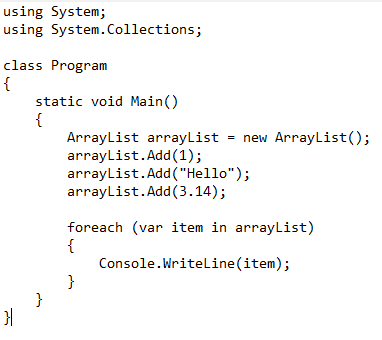
Удобство и функциональность: коллекции обеспечивают удобные методы для управления данными, такие как сортировка, фильтрация, поиск по ключу и т.д.

**Классы коллекций**

В C# существуют различные классы коллекций, которые предназначены для хранения и управления наборами данных. Классы коллекций предоставляют больше возможностей по сравнению с массивами, такие как динамическое изменение размера, возможность добавления и удаления элементов и другие удобные функции. Рассмотрим несколько основных классов коллекций.

**1. ArrayList**

ArrayList — это коллекция, которая может хранить элементы любого типа (object), и её размер может динамически изменяться по мере добавления элементов.



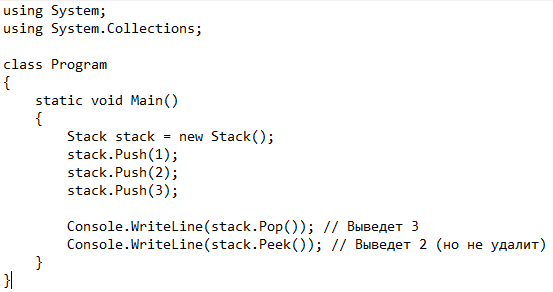
Особенности:

Элементы могут быть любого типа, так как ArrayList работает с типом object.

Изменение размера динамическое.

**2. Stack**

Stack — это коллекция, работающая по принципу "последний вошёл — первый вышел" (LIFO). То есть последний добавленный элемент будет извлечён первым.



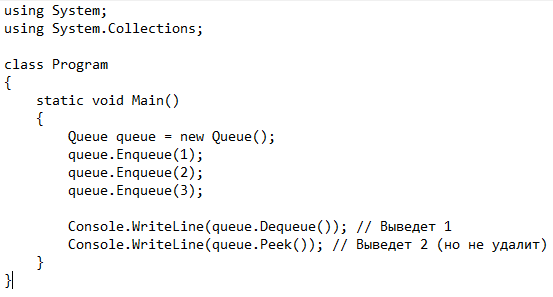
Особенности:

Последний добавленный элемент извлекается первым.

Методы: Push, Pop, Peek.

**3. Queue**

Queue — это коллекция, работающая по принципу "первый вошёл — первый вышел" (FIFO). Элементы добавляются в конец и извлекаются с начала.



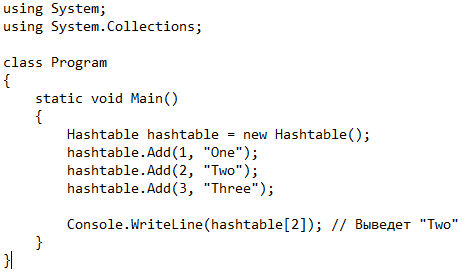
Особенности:

Первый добавленный элемент извлекается первым.

Методы: Enqueue, Dequeue, Peek.

4. Hashtable

Hashtable — это коллекция пар "ключ-значение". Каждый ключ уникален, и по ключу можно получить соответствующее значение.



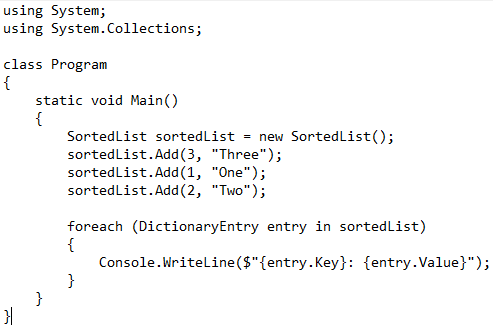
Особенности:

Ключи уникальны.

Работает по принципу хэширования для быстрого доступа к значениям по ключам.

**5. SortedList**

SortedList — это коллекция пар "ключ-значение", которая автоматически сортирует элементы по ключу.



Особенности:

Автоматическая сортировка по ключам.

Быстрый доступ к элементам по ключу, как в Hashtable, но элементы отсортированы.

**Интерфейсы коллекций ICollection, IList, IDictionary, IDictionaryEnumerator**

Интерфейсы в C# являются контрактами, которые определяют набор методов и свойств, которые должны быть реализованы в классах, реализующих их. Интерфейсы коллекций, такие как ICollection, IList, IDictionary, и IDictionaryEnumerator, являются частью системы коллекций в .NET и позволяют работать с различными типами данных.

1. ICollection

ICollection является базовым интерфейсом для всех коллекций. Он предоставляет основные свойства и методы для работы с коллекциями, такие как подсчет элементов и перечисление.

Основные члены ICollection:

- Count: возвращает количество элементов в коллекции.

- IsReadOnly: определяет, является ли коллекция доступной только для чтения.

- Add(T item): добавляет элемент в коллекцию.

- Remove(T item): удаляет элемент из коллекции.

- Contains(T item): проверяет, содержит ли коллекция элемент.

- Clear(): очищает коллекцию.

- CopyTo(T[] array, int index): копирует элементы коллекции в массив, начиная с определенного индекса. (ПРИМЕР 1)

2. IList

IList расширяет интерфейс ICollection и добавляет поддержку индексированной коллекции (то есть коллекции, доступной по индексу).

Основные члены IList:

this[int index]: доступ к элементу по индексу.

Insert(int index, T item): вставляет элемент по определенному индексу.

RemoveAt(int index): удаляет элемент по индексу. (ПРИМЕР 2)

3. IDictionary

IDictionary расширяет ICollection и предоставляет методы и свойства для работы с коллекциями, представляющими пары "ключ-значение".

Основные члены IDictionary:

this[TKey key]: доступ к значению по ключу.

Add(TKey key, TValue value): добавляет элемент (ключ-значение).

Remove(TKey key): удаляет элемент по ключу.

ContainsKey(TKey key): проверяет, содержит ли коллекция элемент с определенным ключом.

Keys: коллекция всех ключей.

Values: коллекция всех значений. (ПРИМЕР 3).

4. IDictionaryEnumerator

IDictionaryEnumerator является специальным интерфейсом для перебора коллекций IDictionary. Он предоставляет способ доступа к элементам как к парам "ключ-значение".

Основные члены IDictionaryEnumerator:

Key: возвращает ключ текущего элемента.

Value: возвращает значение текущего элемента.

Entry: возвращает структуру DictionaryEntry, содержащую как ключ, так и значение текущего элемента.

Все члены из интерфейса IEnumerator (например, MoveNext, Reset, Current). (ПРИМЕР 4).

Обобщенные и необобщенные классы

Классы в C# могут быть обобщенными (generic) и необобщенными (non-generic). Основное различие между ними заключается в том, как они обрабатывают типы данных.

1. Необобщенные классы

Необобщенные классы предназначены для работы с конкретными типами данных. Они жестко привязаны к определенным типам, и для каждого типа данных может потребоваться создание отдельной версии класса, если нужно работать с разными типами данных. (ПРИМЕР 5).

В данном примере класс Box работает только с целыми числами (int). Если нам нужно создать аналогичный класс для работы с другими типами данных (например, с double или string), нам придется создавать отдельные классы для каждого типа.

2. Обобщенные классы

Обобщенные (generic) классы позволяют работать с любыми типами данных без необходимости создавать отдельные версии класса для каждого типа. В этом случае тип данных определяется при создании экземпляра класса. (ПРИМЕР 6).

Здесь обобщённый класс Box<T> может принимать любой тип данных, указанный в угловых скобках при создании экземпляра класса. Например, Box<int> работает с целыми числами, а Box<string> — со строками.

Преимущества обобщенных классов:

Универсальность: один обобщенный класс может работать с различными типами данных, что избавляет от необходимости создавать несколько реализаций.

Безопасность типов: обобщенные классы предоставляют возможность проверять типы данных на этапе компиляции, что снижает вероятность ошибок времени выполнения.

Производительность: использование обобщений снижает необходимость в коробочном преобразовании (boxing) и снижает накладные расходы на работу с объектами типа object.

3. Ограничения (Constraints) в обобщениях

Обобщенные классы могут иметь ограничения на типы данных, с которыми они могут работать. Это помогает убедиться, что используемый тип обладает определёнными свойствами или методами. (ПРИМЕР 7).

В этом примере ограничение where T : struct заставляет параметр T быть типом-значением, таким как int или double. Это предотвращает использование ссылочных типов, таких как string, что гарантирует правильную работу класса с нужным типом данных.

**Делегаты в С#**

Делегаты в C# — это тип, который представляет ссылку на метод. Делегаты позволяют передавать методы как параметры другим методам и определять обратные вызовы. Они также полезны для разработки событий и многопоточных приложений.

Основные моменты:

Объявление делегата: Делегат объявляется с указанием возвращаемого типа и типов параметров, как и у метода.

Назначение метода делегату: Делегат можно связать с методом, который имеет ту же сигнатуру.

Вызов делегата: Делегат можно вызвать так же, как и метод.

**(Пример 8: Простое использование делегата)**

В этом примере:

Объявляется делегат ShowMessage, который принимает строку.

Метод DisplayMessage соответствует сигнатуре делегата.

Делегат вызывается с разными сообщениями.

**(Пример 9: Делегаты как параметры методов)**

Делегаты могут передаваться как параметры в методы, что позволяет создать гибкие и многоразовые функции.

Здесь Operation — делегат, который принимает два целых числа и возвращает результат. В методе PerformOperation делегат передается как параметр, что позволяет вызывать разные методы (например, сложение или вычитание) в зависимости от переданного делегата.

**(Пример 10: Использование анонимных методов и лямбда-выражений)**

В C# можно использовать анонимные методы и лямбда-выражения вместо явного объявления методов.

В этом примере показано, как можно использовать делегаты с анонимными методами и лямбда-выражениями, что делает код более компактным.

**(Пример 11: Делегаты с многоадресностью (Multicast Delegates))**

Делегаты могут хранить несколько методов и вызывать их последовательно.

В данном примере делегат ShowMessage сначала ссылается на метод ShowEnglish, а затем добавляется метод ShowRussian. Когда делегат вызывается, оба метода выполняются последовательно.

**События в С#**

События в C# представляют собой механизм, позволяющий объектам взаимодействовать между собой, оповещая друг друга о произошедших изменениях или событиях. События используются для организации уведомлений (обратных вызовов) от одного объекта к другому.

Основные элементы событий:

Делегаты: События основаны на делегатах. Делегат определяет сигнатуру методов, которые будут вызываться при возникновении события.

События: Это механизм уведомления. Событие связано с делегатом и позволяет присоединять или отсоединять методы-обработчики.

Подписчики: Методы, которые выполняются при наступлении события.

Публикатор: Объект, который инициирует событие.

**Пример 12: Простое использование событий:**

В этом примере:

Класс Publisher инициирует событие Notify через метод OnNotify.

Класс Subscriber подписывается на событие, и его метод OnNotifyReceived вызывается, когда событие происходит.

**Пример 13: Использование событий с несколькими подписчиками**

Можно подписывать несколько обработчиков на одно событие.

В этом примере на событие Notify подписаны два класса: SubscriberA и SubscriberB. Оба подписчика получают уведомление, когда событие инициируется.

**Пример 14: Отписка от событий**

Вы можете удалить обработчик с помощью оператора -=, если больше не хотите, чтобы определённый метод реагировал на событие.

**Пример 15: Стандартные делегаты EventHandler и EventHandler<T>**

C# предоставляет стандартные делегаты для событий: EventHandler и EventHandler<T>. Эти делегаты упрощают работу с событиями.

В этом примере используется стандартный делегат EventHandler, который всегда принимает два параметра: object sender (отправитель события) и EventArgs (информация о событии).