Тема: «Интерфейсы. Коллекции»

Цель: получить навыки создания и реализации интерфейсов. Изучить стандартные коллекции языка C#.

Время выполнения: 4 часа.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Интерфейс

Интерфейс содержит определения для группы связанных функций, которые должен реализовывать неабстрактный `class` или `struct` . Интерфейс может определять методы `static`, которые должны иметь реализацию. Интерфейс может определять реализацию по умолчанию для членов. Интерфейс не может объявлять данные экземпляра, такие как поля, автоматические реализуемые свойства или события, подобные свойствам.

С помощью интерфейсов можно, например, включить в класс поведение из нескольких источников. Эта возможность очень важна в C#, поскольку этот язык не поддерживает множественное наследование классов. Кроме того, необходимо использовать интерфейс, если требуется имитировать наследование для структур, поскольку они не могут фактически наследоваться от другой структуры или класса.

Интерфейс определяется с помощью ключевого слова, `interface` как показано в следующем примере.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Имя интерфейса должно быть допустимым именем идентификатора C#. По соглашению имена интерфейсов начинаются с заглавной буквы I.

Любой объект (класс или структура), реализующий интерфейс `IEquatable<T>`, должен содержать определение для метода `Equals`, соответствующее сигнатуре, которую задает интерфейс. В результате вы можете быть уверены, что класс, реализующий `IEquatable<T>`, содержит метод `Equals`, с помощью которого экземпляр этого класса может определить, равен ли он другому экземпляру того же класса.

Определение `IEquatable<T>` не предоставляет реализацию для метода `Equals`. Класс или структура может реализовывать несколько интерфейсов, но класс может наследоваться только от одного класса.

Интерфейсы могут содержать методы экземпляра, свойства, события, индексаторы, а также любое сочетание этих четырех типов членов. Интерфейсы могут содержать статические конструкторы, поля, константы или операторы. Начиная с C# 11, элементы интерфейса, которые не являются полями, могут быть `static abstract`. Интерфейс не может содержать поля экземпляров, конструкторы экземпляров или методы завершения. Члены интерфейса по умолчанию являются общедоступными, и вы можете явно указать модификаторы доступа, такие как `public`, `protected`, `internal`, `private`, `protected internal` или `private protected`. Элемент `private` должен иметь реализацию по умолчанию.

Для реализации члена интерфейса соответствующий член реализующего класса должен быть открытым и не статическим, а также иметь такое же имя и сигнатуру, что и член интерфейса.

Примечание

Когда интерфейс объявляет статические члены, тип, реализующий этот интерфейс, также может объявлять статические члены с той же сигнатурой. Они отличаются и однозначно идентифицируются типом, объявляющим элемент. Статический элемент, объявленный в типе, не переопределяет статический элемент, объявленный в интерфейсе.

Класс или структуру, реализующие интерфейс, должны предоставлять реализацию для всех объявленных членов без реализации по умолчанию, предоставляемой интерфейсом. Однако если базовый класс реализует интерфейс, то любой класс, производный от базового класса, наследует эту реализацию.

В следующем примере показана реализация интерфейса `IEquatable<T>`. Реализующий класс `Car` должен предоставлять реализацию метода `Equals`.

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car? car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car?.Make, car?.Model, car?.Year);
    }
```

```
}
```

Свойства и индексаторы класса могут определять дополнительные методы доступа для свойства или индексатора, определенного в интерфейсе. Например, интерфейс может объявлять свойство, имеющее акцессор `get`. Класс, реализующий этот интерфейс, может объявлять это же свойство с обоими акцессорами (`get` и `set`). Однако если свойство или индексатор использует явную реализацию, методы доступа должны совпадать.

Интерфейс может наследоваться от одного или нескольких интерфейсов. Производный интерфейс наследует члены от своих базовых интерфейсов. Класс, реализующий производный интерфейс, должен реализовывать все члены в нем, включая все члены базовых интерфейсов производного интерфейса. Этот класс может быть неявно преобразован в производный интерфейс или любой из его базовых интерфейсов. Класс может включать интерфейс несколько раз через наследуемые базовые классы или через интерфейсы, которые наследуются другими интерфейсами. Однако класс может предоставить реализацию интерфейса только однократно и только если класс объявляет интерфейс как часть определения класса (`class ClassName : InterfaceName`). Если интерфейс наследуется, поскольку наследуется базовый класс, реализующий этот интерфейс, то базовый класс предоставляет реализацию членов этого интерфейса. Но производный класс может повторно реализовать любые члены виртуального интерфейса и не использовать наследованную реализацию. Когда интерфейсы объявляют реализацию метода по умолчанию, любой класс, реализующий этот интерфейс, наследует эту реализацию (для доступа к реализации по умолчанию в члене интерфейса необходимо привести экземпляр класса к типу интерфейса).

Базовый класс также может реализовывать члены интерфейса с помощью виртуальных членов. В таком случае производный класс может изменять поведение интерфейса путем переопределения виртуальных членов.

Интерфейс имеет следующие свойства.

- В версии C# 8.0 и более ранних интерфейс подобен абстрактному базовому классу, содержащему только абстрактные элементы. Класс (или структура), реализующий интерфейс, должен реализовывать все его элементы.
- Начиная с C# 8.0 интерфейс может определять реализации по умолчанию для некоторых или для всех его элементов. Класс или структура, реализующие интерфейс, не должны реализовывать элементы, имеющие реализации по умолчанию.

- Невозможно создать экземпляр интерфейса напрямую. Его члены реализуются любым классом (или структурой), реализующим интерфейс.
- Класс или структура может реализовывать несколько интерфейсов. Класс может наследовать базовому классу и также реализовывать один или несколько интерфейсов.

Коллекции

Среда выполнения .NET предоставляет множество типов коллекций, которые хранят группы связанных объектов и управляют ими. Некоторые типы коллекций, такие как `System.ArraySystem.Span<T>`, и `System.Memory<T>` распознаются в языке C#. Кроме того, интерфейсы, такие как `System.Collections.Generic.IEnumerable<T>` распознаются в языке для перечисления элементов коллекции.

Коллекции предоставляют гибкий способ работы с группами объектов. Вы можете классифицировать различные коллекции по следующим характеристикам:

1. Доступ к элементам: каждая коллекция может быть перечислена для доступа к каждому элементу в порядке. Некоторые коллекции обращаются к элементам по индексу, позиция элемента в упорядоченной коллекции. Наиболее распространенным примером является `System.Collections.Generic.List<T>`. Другие коллекции обращаются к элементам по ключу, где значение связано с одним ключом. Наиболее распространенным примером является `System.Collections.Generic.Dictionary<TKey,TValue>`. Вы выбираете между этими типами коллекций в зависимости от способа доступа к элементам приложения.

2. Профиль производительности. Каждая коллекция имеет различные профили производительности для действий, таких как добавление элемента, поиск элемента или удаление элемента. Вы можете выбрать тип коллекции на основе операций, используемых большинством в приложении.

3. Динамическое увеличение и сжатие: большинство коллекций, поддерживают добавление или удаление элементов динамически.

4. Помимо этих характеристик среда выполнения предоставляет специализированные коллекции, которые препятствуют добавлению или удалению элементов, или изменению элементов коллекции. Другие специализированные коллекции обеспечивают безопасность параллельного доступа в многопоточных приложениях.

Все типы коллекций можно найти в справочнике по API .NET.

Массивы представлены `System.Array` и поддерживают синтаксис на языке C#. Этот синтаксис предоставляет более краткие объявления для переменных массива.

`System.Span<T>ref struct` – это тип, предоставляющий моментальный снимок по последовательности элементов без копирования этих элементов. Компилятор применяет правила безопасности, чтобы убедиться `Span`, что доступ к ней невозможно получить после того, как последовательность, на которую она ссылается, больше не находится в области. Он используется во многих API .NET для повышения производительности. `Memory<T>` обеспечивает аналогичное поведение, если не удается использовать `ref struct` тип.

Начиная с C# 12, все типы коллекций можно инициализировать с помощью выражения `Collection`.

Индексируемые коллекции

Индексируемая коллекция – это коллекция, в которой можно получить доступ к каждому элементу с помощью его индекса. Его индекс – это количество элементов перед ним в последовательности. Таким образом, ссылка на элемент по индексу 0 является первым элементом, индексом 1 является второй и т. д. В этих примерах используется `List<T>` класс. Это наиболее распространенная индексируемая коллекция.

В следующем примере создается и инициализируется список строк, удаляется элемент и добавляется элемент в конец списка. После каждого изменения он выполняет итерацию по строкам с помощью инструкции `foreach` или `for` цикла:

```
// Create a list of strings by using a
// collection initializer.
List<string> salmons = ["chinook", "coho", "pink",
"sockeye"];

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye

// Remove an element from the list by specifying
// the object.
```

```
salmons.Remove("coho");


// Iterate using the index:
for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook pink sockeye

// Add the removed element
salmons.Add("coho");
// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye coho
```

В следующем примере элементы из списка удаляются по индексу. Вместо инструкции `foreach` используется `for` оператор, который выполняет итерацию в порядке убывания. Метод `RemoveAt` приводит к тому, что элементы после удаленного элемента имеют более низкое значение индекса.

```
List<int> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Для типа элементов в `List<T>` можно также определить собственный класс. В приведенном ниже примере класс `Galaxy`, который используется объектом `List<T>`, определен в коде.

```
    private static void IterateThroughList()
    {
        var theGalaxies = new List<Galaxy>
        {
            new (){ Name="Tadpole", MegaLightYears=400},
            new (){ Name="Pinwheel", MegaLightYears=25},
            new (){ Name="Milky Way", MegaLightYears=0},
            new (){ Name="Andromeda", MegaLightYears=3}
        };

        foreach (Galaxy theGalaxy in theGalaxies)
        {
                    Console.WriteLine(theGalaxy.Name  +  "    " +
theGalaxy.MegaLightYears);
        }

        // Output:
        //  Tadpole  400
        //  Pinwheel  25
        //  Milky Way  0
        //  Andromeda  3
    }

    public class Galaxy
    {
        public string Name { get; set; }
        public int MegaLightYears { get; set; }
    }
```

Коллекции пар "ключ-значение"

В этих примерах используется `Dictionary<TKey,TValue>` класс. Это самая распространенная коллекция словарей. Коллекция словарей позволяет получить доступ к элементам в коллекции с помощью ключа каждого элемента. Каждый элемент, добавляемый в словарь, состоит из значения и связанного с ним ключа.

В приведенном ниже примере создается коллекция `Dictionary` и выполняется перебор словаря с помощью оператора `foreach`.

```
    private static void IterateThruDictionary()
```

```
    {
                    Dictionary<string,   Element>   elements   =
BuildDictionary();

        foreach (KeyValuePair<string, Element> kvp in elements)
        {
            Element theElement = kvp.Value;

            Console.WriteLine("key: " + kvp.Key);
             Console.WriteLine("values: " + theElement.Symbol +
" " +
                                        theElement.Name   +   "   "   +
theElement.AtomicNumber);
        }
    }

    public class Element
    {
        public required string Symbol { get; init; }
        public required string Name { get; init; }
        public required int AtomicNumber { get; init; }
    }

    private       static       Dictionary<string,       Element>
BuildDictionary() =>
        new ()
        {
            {"K",
                            new  (){  Symbol="K",  Name="Potassium",
AtomicNumber=19}},
            {"Ca",
                             new  (){  Symbol="Ca",  Name="Calcium",
AtomicNumber=20}},
            {"Sc",
                            new  (){  Symbol="Sc",  Name="Scandium",
AtomicNumber=21}},
            {"Ti",
                            new  (){  Symbol="Ti",  Name="Titanium",
AtomicNumber=22}}
        };
```

В приведенном ниже примере используется метод `ContainsKey` и свойство `Item[]Dictionary` для быстрого поиска элемента по ключу. Свойство `Item` позволяет получить доступ к элементу в коллекции `elements` с помощью кода `elements[symbol]` в C#.

```
if (elements.ContainsKey(symbol) == false)
{
    Console.WriteLine(symbol + " not found");
}
else
{
    Element theElement = elements[symbol];
    Console.WriteLine("found: " + theElement.Name);
}
```

В следующем примере метод используется `TryGetValue` для быстрого поиска элемента по ключу.

```
if (elements.TryGetValue(symbol, out Element? theElement) == false)
    Console.WriteLine(symbol + " not found");
else
    Console.WriteLine("found: + theElement.Name);
```

Индивидуальные задания для лабораторной работы

1. Реализовать интерфейс, который будет расширять предметную область из предыдущих лабораторных работ. Добавить в интерфейс:
   - метод вывода на экран всех полей класса;
   - свойство содержащее название объекта;
   - метод, который делает реверс названия объекта.
2. Добавить в интерфейс для одного из методов реализацию по умолчанию.
3. Реализовать интерфейс всеми классами потомками, которые находятся в самом низу иерархии наследования.
4. Добавить в программу любой класс, который будет реализовывать интерфейс.
5. В главном классе создать метод, который будет создавать экземпляр класса из стандартной библиотеки коллекции, в качестве типа хранимых объектов использовать тип интерфейса.
6. Вывести на экран все элементы коллекции, затем сделать реверс названий объектов и повторить операцию.
7. Для обработки всех ошибочных ситуаций использовать конструкцию try…catch().

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание согласно варианту, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание (листинг кода программы, скриншоты консоли с результатами работы).

3. Вывод о проделанной работе.