**Интерфейсы в C#**

**Понятие интерфейса:**

Представьте себе договор или контракт. В этом договоре прописаны определенные обязательства, которые одна сторона должна выполнить. В программировании на C# интерфейс — это такой же договор, только для классов.

Контракт: Интерфейс определяет набор методов, свойств или других членов, которые класс обязан реализовать. Это как список обещаний, которые класс дает.

Шаблон: Интерфейс задает шаблон поведения, которому должен следовать класс.

Абстрактный тип: Интерфейс сам по себе не может быть создан, он служит только для описания того, как должен выглядеть класс.

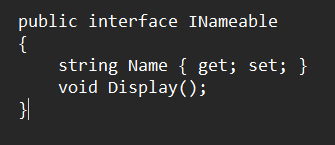
**Зачем нужны интерфейсы:**

Многократное наследование: В отличие от классов, классы могут реализовывать несколько интерфейсов. Это позволяет одному классу иметь поведение из разных источников.

Полиморфизм: Благодаря интерфейсам можно создавать более гибкий и расширяемый код. Например, можно написать метод, который принимает на вход любой объект, реализующий определенный интерфейс.

Отделение интерфейса от реализации: Интерфейс определяет что должен делать класс, а реализация класса определяет как это делать. Это повышает модульность и удобство тестирования кода.

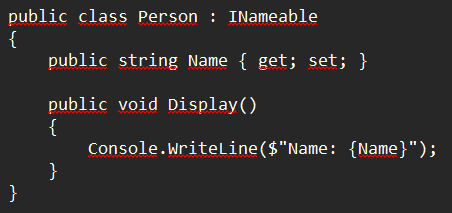
**Синтаксис интерфейса:**



Здесь интерфейс INameable объявляет свойство Name и метод Display, но не предоставляет их реализацию.

**Реализация интерфейса**

Класс, который реализует интерфейс, обязан реализовать все его члены.



В этом примере класс Person реализует интерфейс INameable, предоставляя реализации для свойства Name и метода Display.

Пример использования интерфейса (пример 1);

**Ключевые моменты:**

Объекты интерфейсов: Мы можем создавать массив объектов интерфейса и вызывать их методы через интерфейсный тип.

Полиморфизм: Интерфейсы поддерживают полиморфизм, так как разные классы могут реализовать одинаковый интерфейс по-разному.

**Явная реализация интерфейсов**

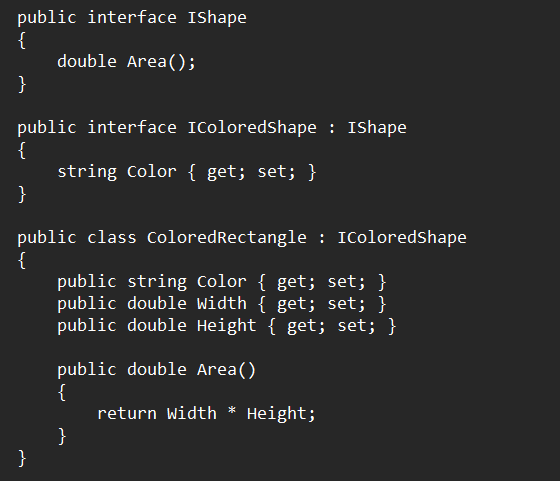
Иногда бывает необходимо реализовать интерфейс, но при этом скрыть его члены от непосредственного использования в классе. Это можно сделать с помощью явной реализации интерфейсов. (пример 2).

Объяснение к примеру 2:

Здесь, метод Print реализован как обычным образом, так и явно для интерфейса IPrintable.

**Наследование интерфейсов**

Интерфейсы могут наследовать другие интерфейсы. Это позволяет создавать более сложные контракты.



Практический пример наследования интерфейсов. (пример 3)

**Обьяснение:**

Наследование интерфейсов: Интерфейс IColoredShape наследует интерфейс IShape, добавляя к нему свойство Color. Класс, который реализует IColoredShape, обязан реализовать как метод Area(), так и свойство Color.

Полиморфизм: Объекты ColoredRectangle и ColoredCircle хранятся как объекты типа IColoredShape. Это демонстрирует мощь полиморфизма — мы можем работать с разными фигурами через общий интерфейс, не зная их конкретные типы.

**Стандартные интерфейсы в C#**

**Интерфейс IEnumerable**

IEnumerable — это один из наиболее часто используемых стандартных интерфейсов в C#. Он определяет механизм, который предоставляет возможность перебора элементов коллекции. Это важный интерфейс, который используется во многих аспектах языка и позволяет коллекциям быть "перечисляемыми" с помощью цикла foreach.

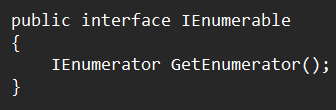
Основные моменты:

Простота: Интерфейс IEnumerable предоставляет только один метод GetEnumerator(), который возвращает объект, реализующий интерфейс IEnumerator.

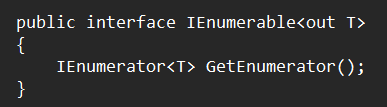
Совместимость с foreach: Коллекции, которые реализуют интерфейс IEnumerable, могут быть использованы в циклах foreach.

Гибкость: Этот интерфейс может быть реализован любым классом, который хочет предоставить возможность перебора элементов, как это делают массивы, списки, словари и другие коллекции.

Сигнатура интерфейса IEnumerable:

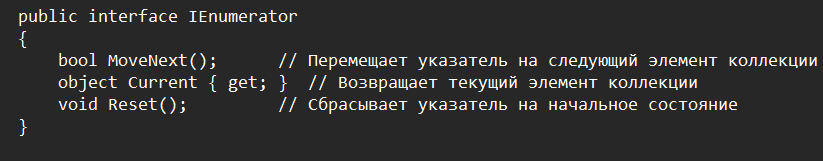


Аналогично, существует обобщённый интерфейс IEnumerable<T>, который предоставляет типобезопасный доступ к элементам коллекции:



**Как работает IEnumerator**

IEnumerator предоставляет методы для перемещения по коллекции и доступ к текущему элементу.



**Пример использования IEnumerable с классом**

Предположим, у нас есть простая коллекция, представляющая хранилище книг, и мы хотим сделать так, чтобы её можно было перебирать с помощью foreach. (пример 4).

Обьяснение:

Класс Book: Представляет объект книги с двумя свойствами: Title и Author.

Класс Library: Реализует интерфейс IEnumerable<Book>, чтобы библиотеку можно было перебирать. Мы используем List<Book> для хранения книг, и метод GetEnumerator просто возвращает перечислитель списка.

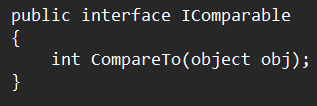
Метод GetEnumerator(): Этот метод возвращает объект, который позволяет последовательно перебирать коллекцию книг.

Использование foreach: Так как библиотека реализует IEnumerable<Book>, мы можем перебирать её элементы с помощью цикла foreach.

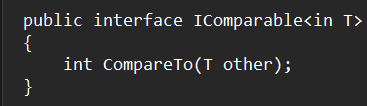
**Интерфейс IComparable**

IComparable — это стандартный интерфейс, который используется для сравнения объектов одного и того же типа. Этот интерфейс определяет метод CompareTo, который возвращает целое число, указывающее порядок объектов относительно друг друга. Интерфейс чаще всего применяется для сортировки объектов в коллекциях, таких как массивы или списки.

Сигнатура интерфейса:



Также существует обобщённый вариант интерфейса IComparable<T>:



Метод CompareTo возвращает следующие значения:

Отрицательное число: Текущий объект меньше, чем объект other.

Ноль: Текущий объект равен объекту other.

Положительное число: Текущий объект больше, чем объект other.

Пример использования IComparable

Рассмотрим класс, который представляет информацию о студентах, и реализуем интерфейс IComparable для сортировки этих объектов по возрасту. (пример 5).

Обьяснение:

**Класс Student:**

Имеет два свойства: Name (имя) и Age (возраст).

Реализует интерфейс IComparable<Student>, чтобы можно было сравнивать студентов по возрасту.

**Метод CompareTo(Student other):**

Сравнивает текущий объект с другим объектом Student.

Используется метод Age.CompareTo(other.Age) для сравнения возраста студентов.

**Использование метода Sort():**

Вызов метода Sort() сортирует список студентов по возрасту благодаря тому, что класс Student реализует интерфейс IComparable<Student>.

**Использование интерфейса IComparable в реальных задачах:**

Сортировка объектов в коллекциях (например, в списках или массивах).

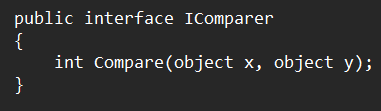
Поиск минимального или максимального элемента в коллекции.

Поддержка логики сравнения объектов в пользовательских классах, например, при работе с деревьями, хэш-таблицами и другими структурами данных.

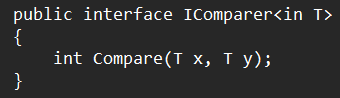
**Интерфейс IComparer**

IComparer — это интерфейс, который позволяет определить собственную логику для сравнения двух объектов. В отличие от интерфейса IComparable, который реализуется в самих объектах для их "естественной" сортировки, IComparer позволяет задать отдельный класс для сравнения объектов. Это удобно, когда нужно использовать несколько различных способов сортировки для одного типа объектов.

Сигнатура интерфейса:



Также есть обобщённая версия интерфейса:



Метод Compare(x, y) возвращает:

Отрицательное число: Если x меньше, чем y.

Ноль: Если x равно y.

Положительное число: Если x больше, чем y.

Этот интерфейс часто используется для задания пользовательской сортировки в таких методах, как List<T>.Sort().

**Пример использования IComparer**

Предположим, у нас есть класс студентов, и мы хотим сортировать их по нескольким критериям: сначала по возрасту, а затем по имени. (пример 6)

**Объяснение кода:**

Класс Student:

Имеет два свойства: Name и Age.

Переопределён метод ToString(), чтобы выводить информацию о студенте в удобном формате.

Класс AgeComparer:

Реализует интерфейс IComparer<Student>, который сравнивает студентов по возрасту.

Метод Compare() сравнивает возраст студентов с помощью метода CompareTo.

Класс NameComparer:

Реализует интерфейс IComparer<Student>, который сравнивает студентов по имени.

Метод Compare() использует статический метод string.Compare() для сравнения строк (имён).

Использование Sort() с переданным IComparer:

Мы можем передать объекты классов AgeComparer или NameComparer в метод Sort(), чтобы сортировать список студентов по нужному критерию.

**Преимущества использования IComparer:**

Гибкость: Позволяет реализовать множество способов сортировки одного и того же объекта, не изменяя сам класс объекта.

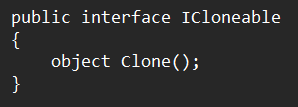
Разделение логики сравнения: Логика сравнения выносится в отдельные классы, что упрощает поддержку и изменение кода.

Модульность: Удобно, если нужно изменять поведение сортировки в зависимости от ситуации, не затрагивая основной класс.

**Интерфейс ICloneable**

Интерфейс ICloneable предоставляет механизм для создания копии объекта. Этот интерфейс содержит один метод Clone(), который возвращает новый объект, являющийся копией текущего объекта. Метод может реализовать поверхностное или глубокое копирование объекта, в зависимости от того, как это реализовано в классе.

Сигнатура интерфейса:



Поверхностное копирование (Shallow Copy): Копируются только значения полей примитивных типов, а ссылки на объекты остаются такими же (объекты не копируются).

Глубокое копирование (Deep Copy): Копируются не только примитивные поля, но и сами объекты, на которые ссылаются поля.

**Пример использования ICloneable**

Рассмотрим класс, представляющий человека, у которого есть поля для имени и адреса. Мы создадим метод Clone, который будет возвращать копию объекта. (пример 7).

Объяснение кода:

Класс Address:

Представляет адрес с полями City (город) и Street (улица).

Метод ToString() переопределён для удобного вывода адреса.

Класс Person:

Имеет поля Name, Age и ссылку на объект Address.

Реализует интерфейс ICloneable и метод Clone().

В методе Clone() выполняется поверхностное копирование с помощью метода MemberwiseClone(), а затем создаётся новая копия объекта Address, чтобы реализовать глубокое копирование.

Метод MemberwiseClone():

Это встроенный метод в C#, который выполняет поверхностное копирование объекта. Он копирует значения всех полей текущего объекта в новый объект, но ссылки на объекты остаются одинаковыми.

Чтобы обеспечить глубокое копирование, нам нужно явно создать новые объекты для полей ссылочного типа (как это сделано для поля Address).

Пример использования:

Создаётся оригинальный объект person1.

Создаётся его клон person2, и мы меняем имя и адрес в клонированном объекте, чтобы показать, что изменения в клоне не влияют на оригинал.

**Когда использовать ICloneable:**

Когда нужно копировать сложные объекты, содержащие как примитивные типы, так и объекты ссылочных типов.

Когда требуется глубокая или поверхностная копия объекта, в зависимости от требований.

Для создания дубликатов объектов без их повторного создания с нуля, особенно если объект содержит сложные структуры данных.

**Коллекция <list>**

В C# коллекция <List> представляет собой динамический массив, который является частью пространства имен System.Collections.Generic. В отличие от обычных массивов, списки могут динамически изменять свой размер, что делает их удобными для работы с неопределённым количеством элементов.

**Основные характеристики <List>:**

* Динамическое изменение размера.
* Универсальность — тип элементов задаётся параметром.
* Включает методы для работы с элементами: добавление, удаление, сортировка, поиск и др.

**Подключение пространства имен**

Для работы с коллекцией <List>, необходимо подключить пространство имен:



Основные методы <List>:

* Add(T item) — добавляет элемент в конец списка.
* Insert(int index, T item) — вставляет элемент на определённую позицию.
* Remove(T item) — удаляет первый встреченный элемент.
* RemoveAt(int index) — удаляет элемент по индексу.
* Clear() — очищает список.
* Count — свойство для получения количества элементов в списке.
* Contains(T item) — проверяет, содержится ли элемент в списке.
* Sort() — сортирует элементы списка.

**Базовое использование <List>**

В этом примере создается список чисел, добавляются элементы, удаляются, а затем выводятся на консоль. (пример 8).

**Работа с объектами в <List>**

В этом примере создается список объектов класса Person, а затем выполняется добавление и вывод данных. (пример 9).

**Сортировка списка**

Здесь демонстрируется сортировка элементов списка с помощью метода Sort(). (пример 10).

**Фильтрация списка с использованием LINQ**

Можно также фильтровать элементы списка с помощью LINQ для более сложных операций: (пример 11).