ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Интерфейсы

- ✓ Проектирование интерфейса
- ✓ Множественная реализация интерфейсов
- ✓ Явная реализация
- ✓ Ковариантность и контрвариантность
- ✓ Generic интерфейсы и их особенность
- ✓ Использование стандартных интерфейсов: IComparable, IEquatable, ICloneable

Интерфейс

Интерфейс (**interface**) представляет собой не более чем просто именованный набор абстрактных элементов. Абстрактные элементы не имеют никакой стандартной реализации.

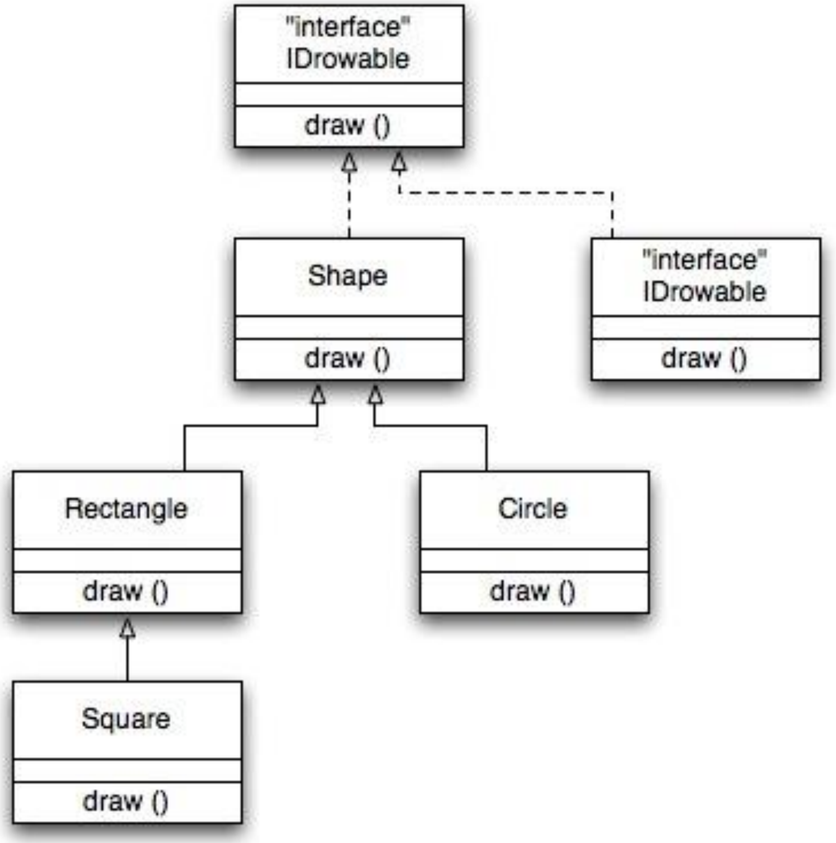
Интерфейс – это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Благодаря поддержке интерфейсов в **C#** может быть в полной мере реализован главный принцип полиморфизма:

Один интерфейс – множество методов.

Интерфейсы могут определять:

- методы;
- свойства;
- индексаторы;
- события;
- статические поля и константы.



Интерфейс

Интерфейс по умолчанию имеет уровень доступа **internal**.

Реализация – через указание имени интерфейса, после «:».

Класс может реализовывать базовый класс и интерфейсы.

В классе должен быть реализован каждый член интерфейса.

Допускается реализация интерфейсов в структуре.

При приведении структуры к интерфейсному типу этот экземпляр будет упакован, потому что интерфейсная переменная является ссылкой, которая должна указывать на объект в куче.

```
interface IMovable
{
    void Move();
}
```

Проектирование интерфейса

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в **C#** начинаются с заглавной буквы **I**, например, **IComparable**, **IEnumerable** (так называемая венгерская нотация), однако это не обязательное требование, а скорее стиль программирования.

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных.

При проектировании интерфейса необходимо следовать принципу разделения интерфейса (**The Interface Segregation Principle**) – этот принцип говорит о недостатках «**жирных**» интерфейсов.

Класс имеет «**жирный**» интерфейс, если функции этого интерфейса недостаточно сцепленные. Иными словами, интерфейс класса можно разбить на группы методов. Каждая группа предназначена для обслуживания разнотипных клиентов. Одним клиентам нужна одна группа методов, другим – другая.

Каждую из подобных группу методов предпочтительно выносить в отдельный интерфейс.

Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в **C#** не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса.

Интерфейсы позволяют частично обойти это ограничение, поскольку в **C#** класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую «**,**».

Допускается наследование интерфейса от интерфейса.

Количество базовых интерфейсов не ограничено.

Количество производных интерфейсов не ограничено.

В классе, который реализует интерфейс унаследованный от других базовых интерфейсов, должны присутствовать все реализации членов интерфейсов в созданной иерархии.

Явная реализация

Кроме неявного применения интерфейсов, существует также явная реализация интерфейса.

При явной реализации указывается название метода или свойства вместе с названием интерфейса, при этом мы не можем использовать модификатор **public**, то есть методы являются закрытыми.

Следует учитывать, что при явной реализации интерфейса его методы и свойства не являются частью класса. Поэтому напрямую через объект класса мы к ним обратиться не сможем.

При отсутствии конфликта сигнатур реализацией интерфейса считается метод подходящий по сигнатруре описанной в интерфейсе.

Реализация множества интерфейсов может иногда приводить к конфликту между сигнатурами членов. Разрешать такие конфликты можно за счет явной реализации члена интерфейса.

При конфликте сигнатур необходимо явно указать какой из методов является реализацией интерфейса.

Ковариантность и контрвариантность

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Имеется три возможных варианта поведения:

- **Ковариантность** – позволяет использовать более конкретный тип, чем заданный изначально. Обобщенные интерфейсы ковариантные, если к универсальному параметру применяется ключевое слово **out**.
- **Контравариантность** – позволяет использовать более универсальный тип, чем заданный изначально. Обобщенные интерфейсы контрвариантные, если к универсальному параметру применяется ключевое слово **in**.
- **Инвариантность** – позволяет использовать только заданный тип. По умолчанию все обобщенные интерфейсы являются инвариантными.

C# позволяет создавать ковариантные и контравариантные обобщенные интерфейсы. Эта функциональность повышает гибкость при использовании обобщенных интерфейсов в программе.

Ковариантность и контрвариантность

При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

```
interface IBank<out T>
{
    T CreateAccount(int sum);
}
```

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к аргументам, используемым в качестве возвращаемых типов.

```
interface ITransaction<in T>
{
    void DoOperation(T account, int sum);
}
```

Generic интерфейсы и их особенность

Термин **обобщение** (**Generic**) – означает параметризированный тип.

Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра.

С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных.

Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

```
interface ITransport<T>
{
    T CreateTransport();
}
```

Generic интерфейсы и их особенность

Использование обощений имеет следующие плюсы:

Производительность – одним из основных преимуществ обобщений является производительность. Использование типов значений с необобщенными классами коллекций вызывает упаковку (**boxing**) и распаковку (**unboxing**), которые являются довольно затратными по производительности операциями.

Безопасность – другим свойством обобщений является безопасность типов. Обобщения автоматически обеспечивают типовую безопасность всех операций. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

"Разбухание" кода – поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в **IL**.

Generic интерфейсы и их особенность

Ограничение на базовый класс позволяет указывать базовый класс, который должен наследоваться аргументом типа.

Ограничение на базовый класс служит двум главным целям:

Оно позволяет использовать в обобщенном классе те члены базового класса, на которые указывает данное ограничение. В отсутствие ограничения на базовый класс компилятору ничего не известно о типе членов, которые может иметь аргумент типа. Накладывая ограничение на базовый класс, вы тем самым даете компилятору знать, что все аргументы типа будут иметь члены, определенные в этом базовом классе.

Ограничение на базовый класс гарантирует использование только тех аргументов типа, которые поддерживают указанный базовый класс. Это означает, что для любого ограничения, накладываемого на базовый класс, аргумент типа должен обозначать сам базовый класс или производный от него класс.

Generic интерфейсы и их особенность

В C# предусмотрен ряд ограничений на типы данных:

- **Ограничение на базовый класс** – требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.
- **Ограничение на интерфейс** – требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- **Ограничение на конструктор** – требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора **new().**
- **Ограничение ссылочного типа** – требует указывать аргумент ссылочного типа с помощью оператора **class**.
- **Ограничение типа значения** – требует указывать аргумент типа значения с помощью оператора **struct**.

```
interface ITransportManufacture<T>
          where T : Transport // ограничение на класс
{
    T CreateTransport();
}
```

Generic интерфейсы и их особенность

Обобщенные интерфейсы имеют следующие особенности:

- **Значения по умолчанию** – при необходимости присвоить переменным универсальных параметров некоторое начальное значение. Но напрямую мы его присвоить не можем. В этом случае нам надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения **null**, а типам значений - значение **0**.
- **Статические поля обобщенных классов** – при типизации обобщенного класса определенным типом будет создаваться свой набор статических членов.
- **Использование нескольких универсальных параметров** – обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять различные типы.

```
class Manufacture<T> : ITransportManufacture<T>
{
    T CreateTransport()
    {
        return default(T);
    }
}
```

Использование стандартных интерфейсов: IComparable

Большинство встроенных в **.NET** классов коллекций и массивы поддерживают сортировку.

С помощью метода, который, как правило, называется **Sort()** можно сразу отсортировать по возрастанию весь набор данных. Однако метод **Sort** по умолчанию работает только для наборов примитивных типов, как **int** или **string**.

Для сортировки наборов сложных объектов применяется интерфейс **IComparable**.

Метод **CompareTo()** предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра. На выходе он возвращает целое число, которое может иметь одно из трех значений:

- **Меньше нуля** – значит, текущий объект должен находиться перед объектом, который передается в качестве параметра.
- **Равен нулю** – значит, оба объекта равны.
- **Больше нуля** – значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра.

Использование стандартных интерфейсов: IEquatable

Интерфейс **IEquatable** служит для определения равенства двух объектов.

Сравниваемый тип данных передается ему в качестве аргумента типа **T**. В этом интерфейсе определяется метод **Equals().**

В методе **Equals()** сравниваются вызывающий объект и другой объект, определяемый параметром **other**. В итоге возвращается логическое значение **true**, если оба объекта равны, а иначе - логическое значение **false**.

В ходе реализации интерфейса **IEquatable<T>** обычно требуется также переопределять методы **GetHashCode()** и **Equals(Object)**, определенные в классе **Object**, чтобы они оказались совместимыми с конкретной реализацией метода **Equals()**.

Использование стандартных интерфейсов: ICloneable

Интерфейс **ICloneable** – это маркерный интерфейс, говорящий другим элементам программы о том, что объекты класса, реализующего этот интерфейс, можно клонировать.

Вызывающий код может не знать, есть ли в этом классе конструктор копирования, а в наличии метода **Clone()** для экземпляра **IClonable** можно не сомневаться.

Помимо этого позволяет создать свою реализацию для более глубокого копирования нежели использование метода **MemberwiseClone()**.

Используется для обхода ограничений которые накладывает использование ссылочных типов данных.

Реализация по-умолчанию C# 8

Реализация по-умолчанию упрощает процесс добавление нового метода в уже существующий интерфейс.

В предыдущих версиях языка такое изменение требовало сразу реализовать новый метод во всех существующих реализациях интерфейса.

C# 8 позволяет определить реализацию по-умолчанию непосредственно в самом интерфейсе.

При этом, во время выполнения используются следующая схема вызова метода интерфейса:

- Если реализация интерфейса содержит вызываемый метод, то будет выполнен код из данной реализации.
- В противном случае будет выполнен код реализации по-умолчанию.

```
interface ITransportManufacture<T> {
    T CreateTransport() {
        return default(T);
    }
}
```

Реализация по-умолчанию C# 8

Проблемы реализации метода по-умолчанию:

Смешение абстракции и реализации

Реализация по-умолчанию является точкой "протечки" реализации в абстракцию. В качестве крайнего примера можно привести сервис, у которого нет необходимости в **private** членах класса для хранения состояния (**stateless**). Значит вся его реализация может быть определена в интерфейсе как “по-умолчанию”.

В результате, интерфейс может легко превратиться, по сути, в базовый абстрактный класс.

Некорректные и неоптимальные реализации

Даже если оставить в стороне вопрос смешивания абстракции и реализации, то существует другая проблема. Реализация по-умолчанию может быть построена на некорректных предположениях о деталях реализаций в классах.

Более того, реализация по-умолчанию может неявно навязывать определенные решения при создании новых реализаций с нуля, являясь подобием базового класса.

Абстрактные классы и Интерфейсы

Абстрактные классы	Интерфейсы
Не могут быть созданы напрямую, но могут содержать конструктор.	Не могут содержать конструктор.
Может хранить данные в полях.	Не может хранить данных . (В C# 8.0 появились статические поля)
Виртуальные элементы могут содержать базовую реализацию. Допустимы невиртуальные элементы.	Все элементы являются виртуальными и не включают реализацию. (C# 8.0 появилась методы по умолчанию)
Класс может наследоваться от единственного абстрактного класса .	Класс может реализовывать несколько интерфейсов.
Класс-наследник может переопределить только некоторые элементы абстрактного класса.	Класс, который реализует интерфейс, должен реализовать все элементы интерфейса.
Наследование поддерживается только для классов.	Интерфейс может быть реализован структурой.

Абстрактные классы и Интерфейсы

Интерфейс описывает только поведение (методы) объекта, а вот состояний (полей) у него нет (кроме **static**), в то время как у абстрактного класса они могут быть.

Абстрактные классы используются, когда есть отношение «наследования», то есть класс-наследник расширяет базовый абстрактный класс, а интерфейсы могут быть реализованы разными классами, вовсе не связанными друг с другом.

Используйте абстрактные классы, если:

- Вы хотите поделиться кодом между несколькими тесно связанными классами.
- Вы ожидаете, что классы, которые расширяют ваш абстрактный класс, имеют много общих методов или полей.
- Вы хотите объявить нестатические поля.

Используйте интерфейсы, если:

- Вы ожидаете, что несвязанные классы будут реализовывать ваш интерфейс.
- Вы хотите определить поведение конкретного типа данных, но вам не важно, кто его реализует.
- Вы хотите использовать множественное наследование типа.

Задание

Создайте базовый класс описывающий транспорт с произвольными полями и методом **Move()** возвращающим способ передвижения в виде строки.

Добавьте производный класс описывающий лодку. Переопределите метод **Move()** чтобы его поведение было актуально.

Добавьте производный класс описывающий автомобиль. Переопределите метод **Move()** чтобы его поведение было актуально.

Добавьте производный класс описывающий Человека. Добавьте скрытое поле типа Транспорт. Создайте метод выводящий информацию о том, как Человек передвигается. Смена типа транспорта возможна исключительно через свойство либо метод.

В методе **Main()** класса **Program** объявите переменную типа транспорт и присвойте экземпляр любого производного класса.

Проверьте поведение вашего кода для различных экземпляров производных классов.