**Практическая работа №9 Работа с объектами через интерфейсы**

Интерфейс – синтаксический и семантический шаблон, которого все классы-наследники должны придерживаться. Интерфейс говорит, что он умеет делать, классы определяют, как они это делают. Интерфейс представляет собой класс без какого-либо кода. Все интерфейсы по умолчанию public, модификатор доступа у интерфейсов не используется. У методов также не используется модификатор доступа, по умолчанию методы public. У методов в интерфейсе не должно быть тела, только заголовки методов.

Определение интерфейса

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

• Методы

• Свойства

• Индексаторы

• События

• Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

interface IMovable {

// константа

const int minSpeed = 0; // минимальная скорость

// статическая переменная

static int maxSpeed = 60; // максимальная скорость

// метод

void Move(); // движение

// свойство

string Name { get; set; } // название

delegate void MoveHandler(string message); // определение делегата для события

// событие

event MoveHandler MoveEvent; // событие движения }

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имеют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

1 static void Main(string[] args)

2 {

3 Console.WriteLine(IMovable.maxSpeed);

4 Console.WriteLine(IMovable.minSpeed);

5 }

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

1 interface IMovable

2 {

3 public const int minSpeed = 0; // минимальная скорость

4 private static int maxSpeed = 60; // максимальная скорость

5 public void Move();

6 protected internal string Name { get; set; } // название

7 public delegate void MoveHandler(string message); // определение делегата для события

8 public event MoveHandler MoveEvent; // событие движения

9 }

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

1 interface IMovable

2 {

3 // реализация метода по умолчанию

4 void Move()

5 {

6 Console.WriteLine("Walking");

7 }

8 }

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

1 interface IMovable

2 {

3 void Move()

4 {

5 Console.WriteLine("Walking"); 6 }

7 // реализация свойства по умолчанию

8 // свойство только для чтения

9 int MaxSpeed { get { return 0; } }

10 }

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным):

1 interface IMovable

2 {

3 public const int minSpeed = 0; // минимальная скорость

4 private static int maxSpeed = 60; // максимальная скорость

5 // находим время, за которое надо пройти расстояние distance со скоростью speed

6 static double GetTime(double distance, double speed) => distance / speed;

7 static int MaxSpeed

8 {

get { return maxSpeed; }

set

{

if (value > 0) maxSpeed = value;

}

} }

class Program

{

static void Main(string[] args)

{

Console.WriteLine(IMovable.MaxSpeed);

IMovable.MaxSpeed = 65;

Console.WriteLine(IMovable.MaxSpeed); double time = IMovable.GetTime(100, 10);

Console.WriteLine(time);

}

}

Модификаторы доступа интерфейсов

Как и классы, интерфейсы по умолчанию имеют уровень доступа internal, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

1 public interface IMovable

2 {

3 void Move();

4 }

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать Add-> New Item... и в диалоговом окне добавления нового компонента выбрать пункт Interface:

Применение интерфейсов

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

1 IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, возьмем следующий интерфейс IMovable:

1 interface IMovable

2 {

3 void Move();

4 }

Затем какой-нибудь класс или структура могут применить данный интерфейс:

1 // применение интерфейса в классе

2 class Person : IMovable

3 {

4 public void Move()

5 {

6 Console.WriteLine("Человек идет");

7 }

8 }

9 // применение интерфейса в структуре

10 struct Car : IMovable

11 {

12 public void Move()

13 {

14 Console.WriteLine("Машина едет");

15 }

16 }

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public. Применение интерфейса в программе:

using System;

namespace HelloApp

{

interface IMovable

{

void Move();

}

class Person : IMovable

{

public void Move()

{

Console.WriteLine("Человек идет");

}

}

struct Car : IMovable

{

public void Move()

{

Console.WriteLine("Машина едет");

}

}

class Program

{

static void Action(IMovable movable)

{

movable.Move();

}

static void Main(string[] args)

{

Person person = new Person();

Car car = new Car();

Action(person);

Action(car);

Console.Read();

}

}

38 }

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод. Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал. Консольный вывод данной программы:

Человек идет Машина едет

Реализация интерфейсов по умолчанию

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Зачем это нужно? Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

class Program

{

static void Main(string[] args)

{

IMovable tom = new Person(); Car tesla = new Car(); tom.Move(); // Walking

tesla.Move(); // Driving

}

}

interface IMovable

{

void Move()

{

Console.WriteLine("Walking");

} }

class Person : IMovable { } class Car : IMovable

{

public void Move()

{

Console.WriteLine("Driving");

} }

В данном случае интерфейс IMovable определяет реализацию по умолчанию для метода Move. Класс Person не реализует этот метод, поэтому он применяет реализацию по умолчанию в отличие от класса Car, который определяет свою реализацию для метода Move.

Стоит отметить, что хотя для объекта класса Person мы можем вызвать метод Move - ведь класс Person применяет интерфейс IMovable, тем не менее мы не можем написать так:

1 Person tom = new Person();

2 tom.Move(); // Ошибка - метод Move не определен в классе Person

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, в отличие, скажем, от языка С++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

1 myClass: myInterface1, myInterface2, myInterface3, ...

2 {

3

4 }

Рассмотрим на примере:

using System;

namespace HelloApp

{

interface IAccount

{

int CurrentSum { get; } // Текущая сумма на счету void Put(int sum); // Положить деньги на счет void Withdraw(int sum); // Взять со счета

}

interface IClient

{

string Name { get; set; }

}

class Client : IAccount, IClient

{

int \_sum; // Переменная для хранения суммы

public string Name { get; set; }

public Client(string name, int sum)

{

Name = name;

\_sum = sum;

}

public int CurrentSum { get { return \_sum; } }

public void Put(int sum) { \_sum += sum; }

public void Withdraw(int sum)

{

if (\_sum >= sum)

{

\_sum -= sum;

}

}

}

class Program

{

static void Main(string[] args)

{

Client client = new Client("Tom", 200); client.Put(30);

Console.WriteLine(client.CurrentSum); //230 client.Withdraw(100);

Console.WriteLine(client.CurrentSum); //130

Console.Read();

}

} }

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обатите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

1 // Все объекты Client являются объектами IAccount

2 IAccount account = new Client("Том", 200);

3 account.Put(200);

4 Console.WriteLine(account.CurrentSum); // 400

5 // Не все объекты IAccount являются объектами Client, необходимо явное приведение 6 Client client = (Client)account;

7 // Интерфейс IAccount не имеет свойства Name, необходимо явное приведение

8 string clientName = ((Client)account).Name;

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IAccount является объектом Client (ведь интерфейс IAccount могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе IAccount, но являются частью класса Client, то нам надо явным образом выполнить преобразование типов: string clientName = ((Client)account).Name;

**Практическая работа №10 Использование стандартных интерфейсов**

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса, которые не имеют реализации по умолчанию. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

1 interface IMovable

2 {

3 void Move();

4 }

5 abstract class Person : IMovable

6 {

7 public abstract void Move();

8 }

9 class Driver : Person

10 {

11 public override void Move()

12 {

13 Console.WriteLine("Шофер ведет машину");

14 }

15 }

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

1 interface IAction

2 {

3 void Move();

4 }

5 class BaseAction

6 {

7 public void Move()

8 {

9 Console.WriteLine("Move in BaseAction");

10 }

11 }

12 class HeroAction : BaseAction, IAction

13 {

14 }

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов: class HeroAction : BaseAction, IAction

Изменение реализации интерфейсов в производных классах

Может сложиться ситуация, что базовый класс реализовал интерфейс, но в классенаследнике необходимо изменить реализацию этого интерфейса. Что в этом случае делать? В этом случае мы можем использовать либо переопределение, либо сокрытие метода или свойства интерфейса.

Первый вариант - переопределение виртуальных/абстрактных методов:

1 interface IAction

2 {

3 void Move();

4 }

5 class BaseAction : IAction

6 {

7 public virtual void Move()

8 {

9 Console.WriteLine("Move in BaseAction");

10 }

11 }

12 class HeroAction : BaseAction

13 {

14 public override void Move()

15 {

16 Console.WriteLine("Move in HeroAction");

17 }

18 }

В базовом классе BaseAction реализованный метод интерфейса определен как виртуальный (можно было бы также сделать его абстрактным), а в производном классе он переопределен.

При вызове метода через переменную интерфейса, если она ссылается на объект производного класса, будет использоваться реализация из производного класса:

1 BaseAction action1 = new HeroAction();

2 action1.Move(); // Move in HeroAction

3

4 IAction action2 = new HeroAction();

5 action2.Move(); // Move in HeroAction

Второй вариант - сокрытие метода в производном классе:

1 interface IAction

2 {

3 void Move();

4 }

5 class BaseAction : IAction

6 {

7 public void Move()

8 {

9 Console.WriteLine("Move in BaseAction");

10 }

11 }

12 class HeroAction : BaseAction

13 {

14 public new void Move()

15 {

16 Console.WriteLine("Move in HeroAction");

17 }

18 }

Также используем эти классы:

1 BaseAction action1 = new HeroAction();

2 action1.Move(); // Move in BaseAction

3

4 IAction action2 = new HeroAction();

5 action2.Move(); // Move in BaseAction

Так как интерфейс реализован именно в классе BaseAction, то через переменную action2 можно обратиться только к реализации метода Move из базового класса BaseAction. Третий вариант - повторная реализация интерфейса в классе-наследнике:

1 interface IAction

2 {

3 void Move();

4 }

5 class BaseAction : IAction

6 {

7 public void Move()

8 {

9 Console.WriteLine("Move in BaseAction");

10 }

11 }

12 class HeroAction : BaseAction, IAction

13 {

14 public new void Move()

15 {

16 Console.WriteLine("Move in HeroAction");

17 }

18 }

В этом случае реализации этого метода из базового класса будет игнорироваться:

1 BaseAction action1 = new HeroAction();

2 action1.Move(); // Move in BaseAction

3

4 IAction action2 = new HeroAction();

5 action2.Move(); // Move in HeroAction

6

7 HeroAction action3 = new HeroAction();

8 action3.Move(); // Move in HeroAction

Также стоит отметить, что в случае с переменной action1 по-прежнему действует ранее связывание, в силу которого через эту переменную можно вызвать реализацию метода Move только из базового класса, который эта переменная представляет. Четвертый вариант: явная реализация интерфейса:

1 interface IAction

2 {

3 void Move();

4 }

5 class BaseAction : IAction

6 {

7 public void Move()

8 {

9 Console.WriteLine("Move in BaseAction");

10 }

11 }

12 class HeroAction : BaseAction, IAction

13 {

14 public new void Move()

15 {

16 Console.WriteLine("Move in HeroAction");

17 }

18 void IAction.Move()

19 {

20 Console.WriteLine("Move in IAction");

21 }

22 }

В этом случае для переменной IAction будет использоваться явная реализация интерфейса IAction, а для переменной HeroAction по прежнему будет использоваться неявная реализация:

1 BaseAction action1 = new HeroAction();

2 action1.Move(); // Move in BaseAction

3

4 IAction action2 = new HeroAction();

5 action2.Move(); // Move in IAction

6

7 HeroAction action3 = new HeroAction();

8 action3.Move(); // Move in HeroAction