# Урок №4. Наследование, интерфейсы

## Содержание

# 1. Наследование в C#

## 1.1. Анализ механизма наследования в C#

Наследование позволяет повторно использовать уже имеющиеся классы, но при этом расширять их возможности. Существует два вида наследования – наследование типа, при котором новый тип получает все свойства и методы родителя и наследование интерфейса, при котором новый тип получает от родителя сигнатуру методов, без их реализации. Все классы, для которых не указан базовый класс, наследуются от класса `System.Object` (краткая форма названия `object`). C# не поддерживает множественного наследования. Это означает, что класс в C# может быть наследником только одного класса, но при этом может реализовывать несколько интерфейсов. Другими словами, класс может наследоваться не более чем от одного базового класса и нескольких интерфейсов.

На следующем рисунке изображен пример диаграммы классов соответственно спецификации UML 2.0. Из диаграммы можно видеть, что классы `Employee`, `Learner`, `Tutor` наследуют класс `Human`. Класс `Learner` является абстрактным классом, а `Tutor` – закрытым. Подробно элементы и их взаимосвязи, изображенные на диаграмме, будут рассмотрены на протяжении урока. (Для изучения визуального моделирования и использования диаграмм UML обратитесь к источникам [1-3]).

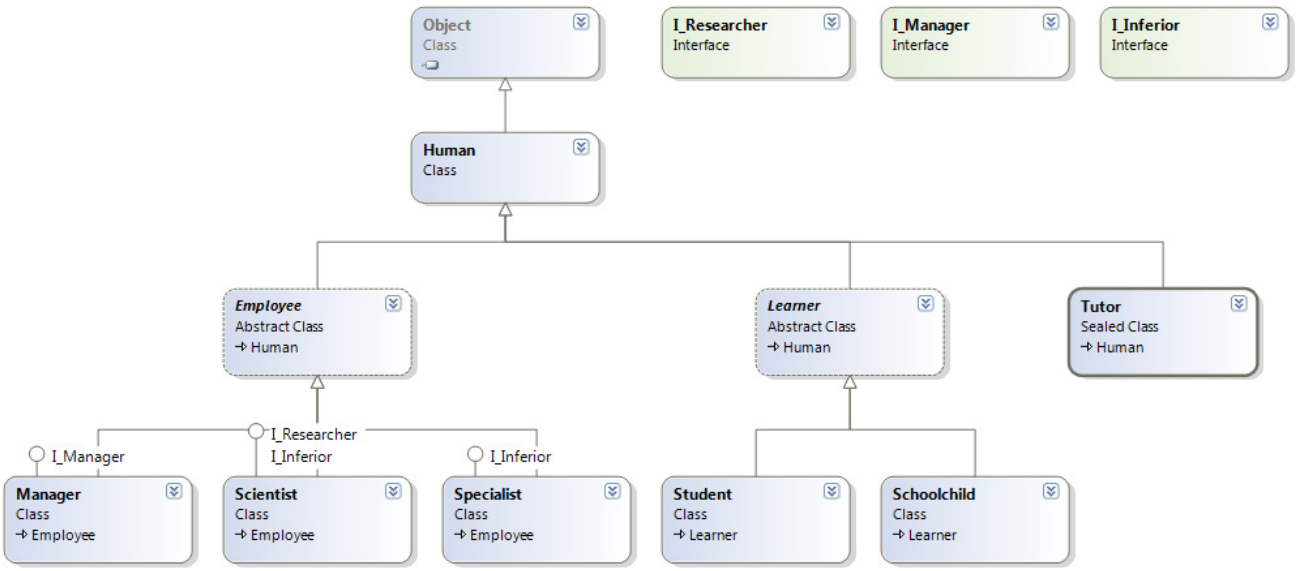


Рис 1. Пример иерархии наследования классов и интерфейсов

### 1.2. Спецификаторы доступа при наследовании

В C# существуют следующие спецификаторы доступа: `private`, `protected`, `public`, `internal`, `protected internal`.

`private` означает, что никакие другие классы не могут получить доступ к методу. Если в базовом классе объявлены private-методы, то наследник не может ими воспользоваться.

`protected` предоставляет доступ только для классов, которые наследуются от данного. Для остальных классов protected-методы недоступны.

`public` делает метод полностью открытым, то есть он является доступным для всех классов.

Ключевое слово `internal` является модификатором доступа для типов и членов типов. Внутренние типы или члены доступны только внутри файлов в одной и той же сборке.

При использовании ключевого слова `protected internal` методы и свойства доступны только в пределах одной сборки, а также для всех производных элементов.

Классам, как и их элементам, может быть назначен любой из этих уровней доступа. Если для элемента класса указан иной модификатор прав доступа, чем для класса,- приоритет у более строгого модификатора.

```
public class Human
{
    private String fName;
    public String FirstName
    {
        get { return fName; }
        set { fName = value; }
    }
}
```

Никакой другой класс не сможет получить доступ к полю `fName` (так как оно имеет модификатор доступа `private`), к свойству `FirstName` получат доступ все классы (потому что оно имеет модификатор доступа `public`).

### 1.3. Особенности использования конструкторов при наследовании

Конструктор наследования в C# имеет следующий вид:

```
class НаследуемыйКласс : БазовыйКласс
{
    // поля, свойства, события и методы класса
```

```
}
```

Если класс наследуется от базового класса и нескольких интерфейсов, то они перечисляются через запятую:

```
class НаследуемыйКласс : БазовыйКласс, Интерфейс1, Интерфейс2
{
    // поля, свойства, события и методы класса
}
```

При создании класса – наследника на самом деле вызывается не один конструктор, а целая цепочка конструкторов. Сначала выбирается конструктор класса, экземпляр которого создается. Этот конструктор пытается обратиться к конструктору своего непосредственного базового класса. Этот конструктор в свою очередь пытается вызвать конструктор своего базового класса. Так происходит, пока не доходим до класса `System.Object`, который не имеет базового класса. В результате имеем последовательный вызов конструкторов всех классов иерархии, начиная с `System.Object` заканчивая классом, экземпляр которого хотим создать. В этом процессе каждый конструктор инициализирует поля собственного класса.

Для каждого класса можно определить несколько конструкторов. Если мы для класса-наследника хотим вызвать конструктор базового класса, то необходимо использовать ключевое слово `base()`.

```
public НаследуемыйКласс() : base()
{
    // поля, свойства, события и методы класса
}
```

Усовершенствуем класс `Human` добавив конструктор не принимающий параметров; конструктор принимающий в качестве параметров имя, отчество и фамилию;  конструктор принимающий в качестве параметров имя, отчество, фамилию и дату рождения:

```
public class Human
{
    protected String fName;
    public String FirstName
    {
        get { return fName; }
        set { fName = value; }
    }

    protected String mName;
    public String MiddleName
    {
```

```
        get { return mName; }
        set { mName = value; }
    }

    protected String lName;
    public String LastName
    {
        get { return lName; }
        set { lName = value; }
    }

    protected DateTime birthday;
    public DateTime Birthday
    {
        get { return birthday; }
        set { birthday = value; }
    }

    public Human() { }

    public Human(String FirstName, String MiddleName, String LastName)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
    }

    public Human(String FirstName, String MiddleName, String LastName,
        DateTime Birthday)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
        this.birthday = Birthday;
    }

    public void Work()
    {
        // Do something
    }
}
```

Тогда класс-наследник может быть реализован следующим образом:

```
public class Employee : Human
{
    public Employee()
        : base()
    {
        // Create Employee object
    }

    public Employee(String FirstName, String MiddleName,
        String LastName) : base(FirstName, MiddleName, LastName) { }
```

```
    public Employee(String FirstName, String MiddleName,
        String LastName, DateTime Birthday)
        : base(FirstName, MiddleName, LastName, Birthday) { }
}
```

### 1.4. Сокрытие имен при наследовании

Сокрытие имен происходит, когда в базовом классе и в классе-наследнике объявлены методы с одинаковым именем. В такой ситуации метод базового класса скрывается и программа может работать не так как предусматривал программист. В таких случаях необходимо воспользоваться модификатором `new`, который скажет компилятору о вашем явном намерении скрыть метод базового класса и использовать метод, объявленный в классе наследнике. Например, пусть у нас есть класс `Human`, который имеет метод `Work()`.

```
class Human
{
    public void Work()
    {
        // Do something
    }
}
```

Объявим еще один класс `Employee`, который наследуется от класса `Human` и объявим в нем метод, который тоже назовем `Work()`. Чтобы в дальнейшем избежать путаницы, воспользуемся модификатором `new`:

```
public class Employee : Human
{
    public new void Work()
    {
        // Do something great
    }
}
```

### 1.5. Ключевое слово base

Ключевое слово `base` используется для доступа к членам базового класса из производного, для вызова метода базового класса при его переопределении в классе наследнике и при создании конструктора класса наследника, который должен вызвать конструктор класса родителя. Доступ к базовому классу разрешен только в конструкторе, методе экземпляра или методе доступа экземпляра.

Вернемся к предыдущему примеру. Предположим, что мы хотим вызвать в классе `Employee` метод `Work()` определенный в базовом классе `Human`:

```
public class Employee : Human
{
    public Employee() : base()
    {
        // Create Employee object
    }

    public override void Work()
    {
        base.Work();
        // Do something great
    }
}
```

# 2. Наследование исключений

## 2.1. Наследование и исключения

Исключение или исключительная ситуация – это ошибка, которая возникает при выполнении программы. Часто они возникают по вине пользователя, например, при вводе данных. Для предотвращения таких ситуаций, программист может проверить допустимость вводимых данных и написать соответствующие обработчики. Но могут возникнуть непредвиденные ситуации, например, нехватка памяти. Полностью предотвратить исключения невозможно, но можно сделать так, чтобы программа в ошибочных ситуациях не заканчивала свою работу аварийно, а вызывала метод, который обработает должным образом эту ситуацию. (Подробно исключения рассмотрены в уроке №3). Обработчик исключения – фрагмент кода, который выполняется при возникновении определенной ошибки.

В C# исключения представлены в виде класса. Программист имеет возможность воспользоваться встроенными исключениями, или создавать свои. Все исключения в C# наследуются от класса `System.Exception`. Из этого класса выведены два класса исключений: `SystemException` (исключения, которые генерируются общеязыковой средой выполнения CLR) и `ApplicationException` (генерируются прикладными программами).

## 2.2. Наследование от стандартных классов исключений

При создании исключений в своих программах программист наследует их от `ApplicationException`. Классы-наследники должны иметь как минимум четыре конструктора: один по умолчанию, второй, задающий свойство сообщению, третий, задающий свойства `Message` и `InnerException`, четвертый – для сериализации исключения, по-

скольку новые классы исключений должны быть сериализируемые. (Подробно сериализация будет рассмотрена в следующих уроках).

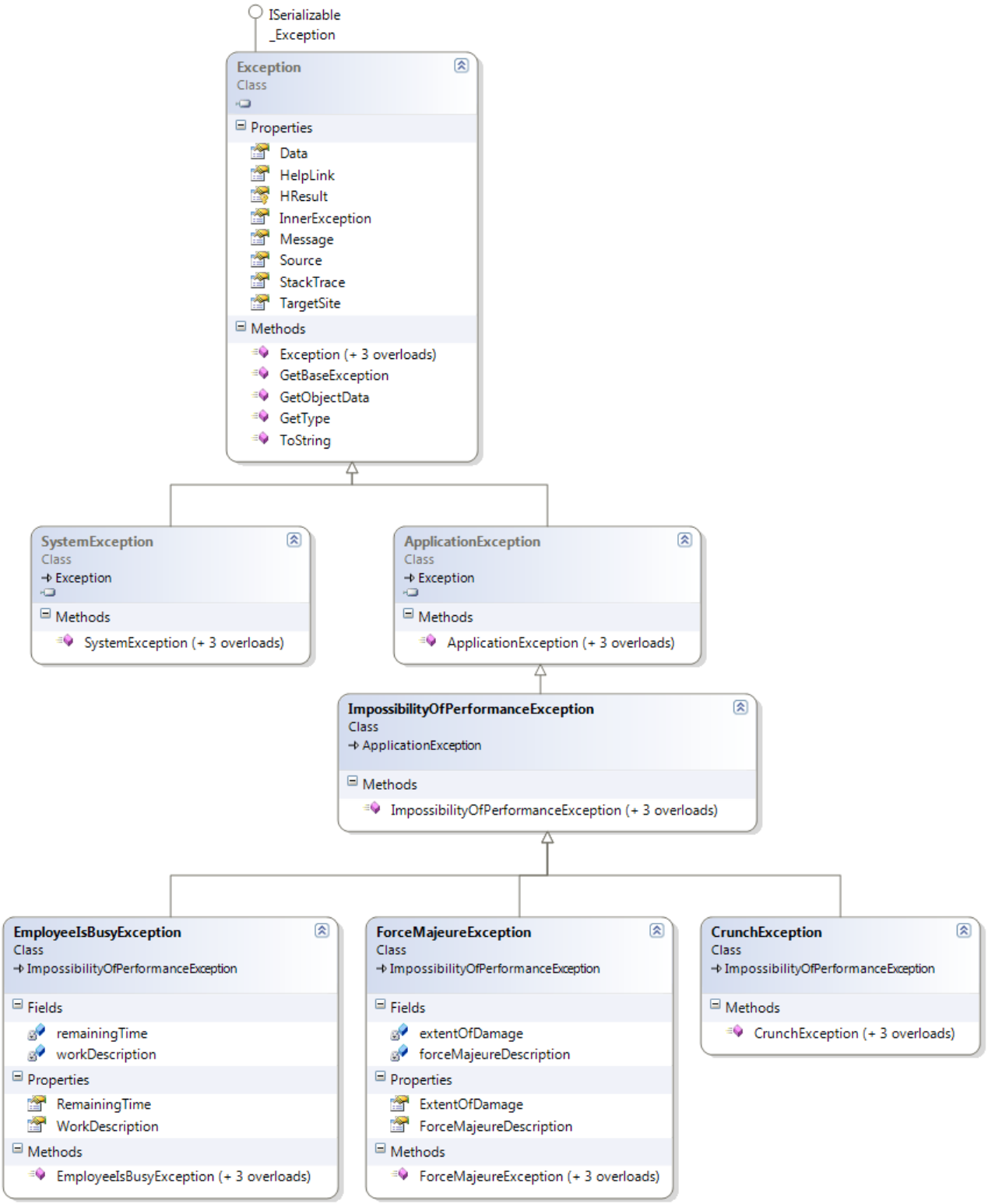


Рис 2. Пользовательские классы исключений должны наследоваться от класса `ApplicationException`

Для примера рассмотрим классы исключений невозможности выполнить работу служащим. Базовым классом для генерации исключения является класс `ImpossibilityOfPerformanceException`:

```
class ImpossibilityOfPerformanceException : ApplicationException
{
    public ImpossibilityOfPerformanceException()
        : base() { }

    public ImpossibilityOfPerformanceException(String message)
        : base(message) { }

    public ImpossibilityOfPerformanceException(String message,
        Exception innerException)
        : base(message, innerException) { }

    public ImpossibilityOfPerformanceException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Класс `ImpossibilityOfPerformanceException` наследуют три класса:

1. Класс исключения, что служащий не может выполнить работу в связи с тем, что он занят выполнением другой работы: `EmployeeIsBusyException`.

```
public class EmployeeIsBusyException
    : ImpossibilityOfPerformanceException
{
    public EmployeeIsBusyException()
        : base() { }

    public EmployeeIsBusyException(String message)
        : base(message) { }

    public EmployeeIsBusyException(String message,
        Exception innerException)
        : base(message, innerException) { }

    public EmployeeIsBusyException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }

    private DateTime remainingTime;
    public DateTime RemainingTime
    {
        get { return remainingTime; }
        set { remainingTime = value; }
    }

    private String workDescription;
```

```
    public String WorkDescription
    {
        get { return workDescription; }
        set { workDescription = value; }
    }
}
```

2. Класс исключения невозможности выполнить работу по причине форс-мажорных обстоятельств: `ForceMajeureException`.

```
public class ForceMajeureException
    : ImpossibilityOfPerformanceException
{
    public ForceMajeureException()
        : base() { }

    public ForceMajeureException(String message)
        : base(message) { }

    public ForceMajeureException(String message,
        Exception innerException)
        : base(message, innerException) { }

    public ForceMajeureException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }

    private String forceMajeureDescription;
    public String ForceMajeureDescription
    {
        get { return forceMajeureDescription; }
        set { forceMajeureDescription = value; }
    }

    private Decimal extentOfDamage;
    public Decimal ExtentOfDamage
    {
        get { return extentOfDamage; }
        set { extentOfDamage = value; }
    }
}
```

3. Класс исключения невозможности выполнить работу по причине нехватки ресурсов или поломки инструментов: `CrunchException`.

```
public class CrunchException : ImpossibilityOfPerformanceException
{
    public CrunchException()
        : base() { }
```

```
    public CrunchException(String message)
        : base(message) { }

    public CrunchException(String message,
        Exception innerException)
        : base(message, innerException) { }

    public CrunchException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Используя вышеописанные классы исключений для класса `Employee` метод `Work()` можно переопределить следующим образом:

```
public override void Work()
{
    if (this.isWorking)
    {
        throw new EmployeeIsBusyException("Employee is busy");
    }
    else
        if (VerificationResources())
            try
            {
                this.isWorking = true;

                base.Work();

                // Do something

                this.isWorking = false;
            }
            catch (Exception exception)
            {
                throw new ForceMajeureException(
                    "Force-majeure exception", exception);
            }
        else
        {
            throw new CrunchException(
                "Verification resources failed");
        }
}

private Boolean VerificationResources()
{
    Boolean verificationResult = true;
    // verification
    return verificationResult;
}
```

## 3. Использование ключевого слова sealed

Иногда возникают ситуации, когда необходимо запретить наследовать некоторый класс или переопределять некоторый метод. Для этого класс или метод нужно объявить как терминальный. Для этого используют ключевое слово `sealed`. Объявим терминальный класс:

```
public sealed class Tutor : Human { }
```

Никакие другие классы не могут наследовать класс `Tutor`. При попытке это сделать компилятор выдаст ошибку. При этом сам класс `Tutor` может быть наследован от другого класса.

Аналогично можно запретить переопределять свойства и методы базового класса. Если член базового класса имеет модификатор наследования `virtual` (подробнее ключевое слово `virtual` будет рассмотрено ниже в пункте 4.), тогда наследник может закрыть дальнейшее переопределение члена. Разрешим переопределение наследуемыми классами метода `Work()` класса `Human`.

```
class Human
{
    public virtual void Work()
    {
        // Do something
    }
}
```

Тогда класс `Employee` может переопределить метод `Work()` и закрыть возможность его переопределения для собственных наследников.

```
public class Employee : Human
{
    public sealed override void Work()
    {
        // Do something great
    }
}
```

При попытке откомпилировать следующий код, компилятор выдаст ошибку: `Cannot override inherited member 'Employee.Work()' because it is sealed`

```
public class Manager : Employee
{
    public override void Work()
    {
        // Try to do something unbelievable
    }
}
```

# 4. Использование ссылок на базовый класс

Как известно, в C# нельзя присвоить переменной одного типа значение переменной другого типа. Но существует одно исключение - ссылочную переменную базового класса можно присвоить ссылке на объект любого класса-наследника этого базового класса. Когда ссылка на производный класс присваивается ссылочной переменной базового класса, вы получаете доступ только к тем частям объекта, которые определены в базовом классе. Важность такого присвоения ощущается, когда в иерархии классов вызываются конструкторы. Можно определить конструктор, который в качестве параметра принимает объект своего класса. Это позволяет классу создавать копию объекта.

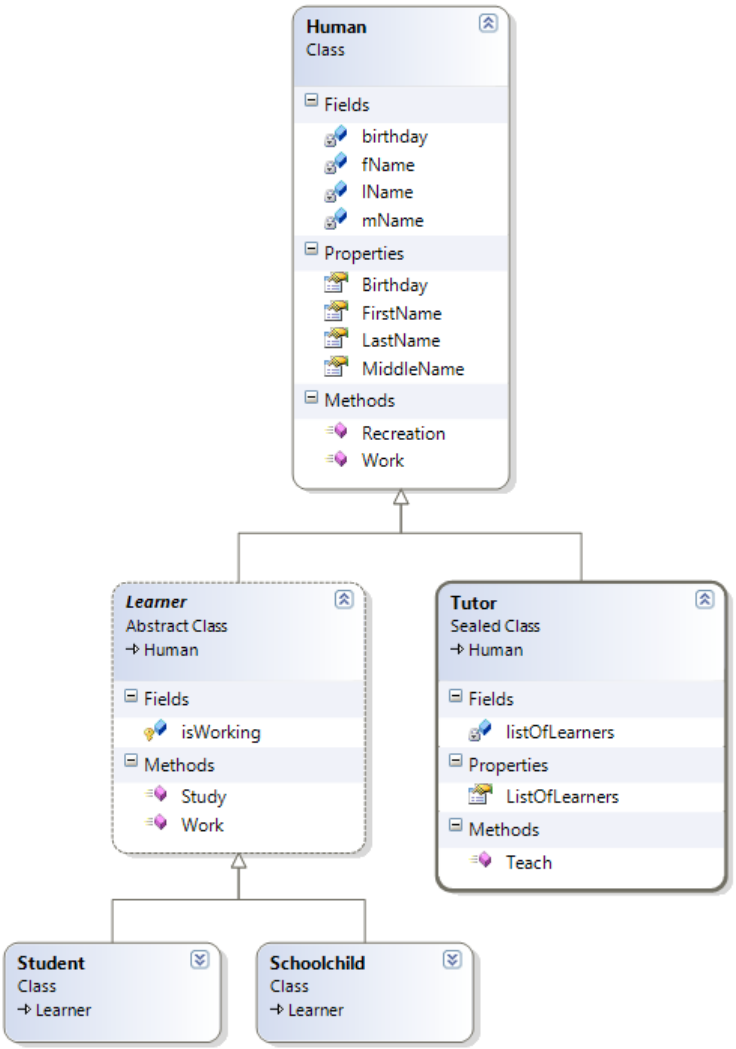


Рис. 3. Диаграмма иерархического взаимодействия

Покажем это на следующем примере. Класс `Tutor`, имеет свойство `ListOfLearners`, `Tutor` обучает объекты типов `Student` и `Schoolchild`, но список является списком объ-

ектов типа `Learner`. Так как `Student` и `Schoolchild` наследуют `Learner` то в список `ListOfLearners` правомерно добавление объектов этих классов.

Следующий код создает объекты типов `Student` и `Schoolchild` и заносит их в список учеников тренера `ListOfLearners`.

```
void CreateTutorTeam(Tutor tutor)
{
    // Создаем команду учеников
    Student student1 = new Student();
    Student student2 = new Student();
    Student student3 = new Student();
    Student student4 = new Student();
    Schoolchild schoolchild1 = new Schoolchild();
    Schoolchild schoolchild2 = new Schoolchild();

    student1.FirstName = "Иван";
    student2.FirstName = "Петр";
    student3.FirstName = "Сергей";
    student4.FirstName = "Максим";
    schoolchild1.FirstName = "Дима";
    schoolchild2.FirstName = "Саша";

    // Заносим учеников в список
    tutor.ListOfLearners.Add(student1);
    tutor.ListOfLearners.Add(student2);
    tutor.ListOfLearners.Add(student3);
    tutor.ListOfLearners.Add(student4);
    tutor.ListOfLearners.Add(schoolchild1);
    tutor.ListOfLearners.Add(schoolchild2);

    // Дальше можно работать со списком ListOfLearners
    // как со списком объектов типа Learner
}
```

Код класса `Tutor`:

```
public sealed class Tutor : Human
{
    public void Teach()
    {
        // Do something
    }

    private List<Learner> listOfLearners;
    public List<Learner> ListOfLearners
    {
        get { return listOfLearners; }
    }
}
```

# 5. Виртуальные методы

## 5.1. Что такое виртуальный метод

Иногда необходимо изменить методы, которые наследуются от базового класса. Для того чтобы была возможность его изменить используют виртуальные методы. Объявив метод или свойство класса как виртуальные, вы тем самым позволяете классам наследникам переопределять данный метод или свойство. Для этого используется ключевое слово `virtual`. Так в классе `Human` виртуальный метод `Work()`, каждый из классов-наследников `Manager`, `Tutor`, `Student` и т.д. может переопределить соответственно своих функций.

- Поля-члены и статические методы не могут быть объявлены как виртуальные.
- Применение виртуальных методов позволяет реализовывать механизм позднего связывания.
- На этапе компиляции строится только таблица виртуальных методов, а конкретный адрес метода, который будет вызван, определяется на этапе выполнения.

При вызове метода - члена класса действуют следующие правила:

- для виртуального метода вызывается метод, соответствующий типу объекта, на который имеется ссылка;
- для не виртуального метода вызывается метод, соответствующий типу самой ссылки.

При позднем связывании определение вызываемого метода происходит на этапе выполнения (а не при компиляции) в зависимости от типа объекта, для которого вызывается виртуальный метод.

## 5.2. Необходимость использования виртуальных методов

Виртуальные методы необходимы, когда классу-наследнику нужно изменить некоторые методы, определенные в базовом классе. Виртуальные методы позволяют определить базовому классу методы, реализация которых есть общей для всех производных классов, и методы, которые можно переопределять. Это позволяет поддерживать динамический полиморфизм. Определения классов-наследников собственных методов становится более гибким, по-прежнему оставляя в силе требование согласующегося интерфейса.

### 5.3. Переопределение виртуальных методов

Для того чтобы переопределить метод в классе-наследнике, используют ключевое слово `override`, так в следующем примере абстрактный класс `Learner` переопределяет метод `Work()` базового класса `Human` (абстрактный класс – это класс, экземпляр которого нельзя создать. Более детально это будет рассмотрено в следующем пункте.)

```
public abstract class Learner : Human
{
    public event EventHandler WorkEnded;

    public void Study()
    {
        // Studing
    }

    public override void Work()
    {
        this.Study();
    }
}
```

## 6. Абстрактный класс

C# позволяет использовать абстрактные классы и методы. Абстрактный класс – это класс, экземпляры которого создавать нельзя, но его могут наследовать другие классы. Для объявления такого класса используют ключевое слово `abstract`. Так в предыдущем примере класс `Learner` является абстрактным.

Если попытаться создать экземпляр класса `Learner`, то компилятор выдаст ошибку: `Cannot create an instance of the abstract class or interface 'Learner'.`

Абстрактный класс будет полезен, если несколько классов имеют общие свойства и методы.

Рассмотрим классы `Student` и `Schoolchild`. Студенты и ученики во время обучения являются подчиненными для тренера, соответственно должны реализовывать интерфейс `I_Inferior`, но реализация его элементов для данных классов не отличается и поэтому практичнее реализовать этот интерфейс в базовом абстрактном классе `Learner`. Аналогично свойство `SubjectList` также следует реализовать в базовом абстрактном классе:

```
public abstract class Learner : Human , I_Inferior
{
    protected Boolean isWorking = false;

    public virtual void Study()
```

```
    {
        // Studing
    }

    public override void Work()
    {
        this.isWorking = true;

        this.Study();

        this.isWorking = false;
    }

    #region I_Inferior Members
        event EventHandler I_Inferior.WorkEnded
        {
            add { throw new NotImplementedException(); }
            remove { throw new NotImplementedException(); }
        }

        bool I_Inferior.IsWorking
        {
            get { return isWorking; }
        }

        object I_Inferior.Work()
        {
            this.isWorking = true;

            object result = new object();

            this.Study();

            // Make result

            this.isWorking = false;
            return result;
        }
    #endregion

    protected List<String> subjectList;
    public List<String> SubjectList
    {
        get { return subjectList; }
    }
}
```

При необходимости классы-наследники могут переопределить или закрыть наследуемые элементы базового абстрактного класса.

## 7. Анализ базового класса Object

Так как каждый тип в C# наследует класс `object`, то его методы будут доступны для любого класса.

Рассмотрим методы класса `object`.

| Метод | Назначение |
|---|---|
| `public virtual bool Equals (object ob1)` | В качестве параметра получает объект типа `object`. Возвращает `true`, если объект, вызывающий метод, и объект, передаваемый в качестве параметра одинаковые. Иначе возвращает `false`. |
| `public static bool Equals (object ob1, object ob2)` | В качестве параметров получает 2 объекта типа `object`. Возвращает `true`, если объекты одинаковые. Иначе возвращает `false`. |
| `protected Finalyze()` | Действует как сборщик мусора. Этот метод используют для очистки ресурсов, которые занимает ссылочный тип. |
| `public virtual int GetHashCode()` | Возвращает хэш-код вызывающего объекта. |
| `public Type GetType()` | Возвращает тип вызывающего объекта. |
| `protected object MemberwiseClone()` | Создает копию вызывающего объекта, в которую копируются все члены класса, но при этом не копируются объекты, на которые ссылаются эти члены. |
| `public static bool ReferenceEquals(object ob1, object ob2)` | Возвращает `true`, если объекты `ob1` и `ob2` (которые передаются в качестве параметров) ссылаются на один и тот же объект. Если они ссылаются на разные объекты, то возвращает `false`. |
| `public virtual string ToString()` | Вызывает строку, которая описывает вызывающий объект. |

## 8. Упаковка, распаковка (boxing, unboxing)

Как уже упоминалось, в C# любой класс (тип) наследует базовый тип `object`. Это значит, что любой тип можно преобразовать к типу `object` и обратно. После присваивания переменной `ob1` типа `object` переменную `x` другого типа, перемененная `ob1` будет хранить в себе значение переменной `x`. Этот процесс называют упаковка (boxing). То есть, значение `x` как бы упаковывается в переменную `ob1`. Этот процесс происходит неявно, достаточно лишь присвоить переменной `ob1` типа `object` переменную `x` другого типа.

Распаковка (unboxing) происходит при извлечении из `ob1` значения переменной `x`. Здесь нужно использовать явное преобразование типов. То есть, чтобы совершить процесс распаковки переменной `ob1` необходимо:

- точно знать тип упакованной переменной;
- создать переменную необходимого типа и присвоить ей приведенное к этому типу значение переменной `ob1`.

Покажем на примере упаковку и распаковку.

```
public void Box_and_Unbox()
{
    object ob1;
    int x = 12;
    ob1 = x; //происходит упаковка значения переменной x в ob1
    int a = (int)ob1; // происходит распаковка, используется
    // явное преобразование типов.
}
```

# 9. Интерфейсы

## 9.1. Понятие интерфейса

Интерфейс содержит описание свойств, методов или событий реализацию которых обязуется осуществить класс-наследник интерфейса. Нельзя прописывать реализацию членов в самом интерфейсе. Также нельзя создать экземпляр интерфейса. Интерфейс не может содержать поля или перегрузки операторов. Все методы интерфейса по умолчанию являются `public`, их нельзя объявить как `virtual` или `static` (изменить модификатор доступа можно при реализации метода в классе-наследнике). После объявления интерфейса его может наследовать любое количество классов. Каждый класс может наследовать любое количество интерфейсов. Но если класс реализовывает интерфейс, он должен реализовать все члены интерфейса. Причем разные классы могут по-разному реализовывать один и тот же интерфейс.

Наследование и реализация интерфейса дает возможность, другим классам определив, что объект реализует некоторый интерфейс, быть абсолютно уверенными, что данный объект реализовывает все элементы описанные сигнатурой интерфейса, так как проверка реализации классами-наследниками элементов интерфейса происходит на этапе компиляции. Таким образом, интерфейсы инкапсулируют свойства, методы и события, описывая только их сигнатуру, и предоставляя их для внешних классов; и требуют реализацию данных свойств, методов и событий от классов-наследников.

### 9.2. Синтаксис объявления интерфейсов

Объявление интерфейса осуществляется с помощью служебного слова `interface` и имеет следующий вид:

```
[модификатор доступа] interface имя_интерфейса
{
    // члены интерфейса
}
```

Название интерфейса принято начинать с заглавной буквы I. Например, основной работой исследователя является исследования и изобретения, тогда интерфейс исследователя можно описать следующим образом:

```
public interface I_Researcher
{
    void Investigate();
    void Invent();
}
```

### 9.3. Примеры создания интерфейсов

Рассмотрим пример интерфейса `I_Inferior`.

```
public interface I_Inferior
{
    event EventHandler WorkEnded;

    Boolean IsWorking { get; }

    object Work();
}
```

Данный интерфейс `I_Inferior` описывает подчиненного служащего. Соответственно приведенному интерфейсу каждый класс, который реализует интерфейс, должен реализовать метод `Work()` описывающий выполнение некоторой работы, специфической для каждого класса; свойство `IsWorking` указывающее занят ли служащий; и событие `WorkEnded` оповещающее о завершении выполнения работы.

Следующий пример интерфейса определяет, что каждый класс реализующий интерфейс `I_Manager` должен иметь список подчиненных `ListOfInferiors` и реализацию методов `Plan(), Organize(), Motivate(), MakeBudget(), и Control()`.

```
public interface I_Manager
{
    List<I_Inferior> ListOfInferiors { get; }

    void Plan();

    void Organize();

    void Motivate();
```

```
    void MakeBudget();

    void Control();
}
```

### 9.4. Интерфейсные ссылки

В C# можно создавать интерфейсную переменную ссылки. Она может указывать на любой экземпляр любого класса, который наследует данный интерфейс. При вызове метода объекта с помощью ссылки выполняется та версия метода, которая реализовывается в этом объекте. Через интерфейсные ссылки можно вызывать только методы данного интерфейса. Если понадобится вызвать метод, который не является частью данного интерфейса, то необходимо привести ссылку к соответствующему типу. В следующем примере рассмотрено создание списка подчиненных:

```
public void CreateListOfInferiors()
{
    I_Manager manager = new Manager();

    Scientist scientist1 = new Scientist();
    scientist1.FirstName = "Bill";

    Specialist specilist1 = new Specialist();
    specilist1.FirstName = "Saimon";

    Specialist specilist2 = new Specialist();
    specilist2.FirstName = "Andrew";

    manager.ListOfInferiors.Add(scientist1);
    manager.ListOfInferiors.Add(specilist1);
    manager.ListOfInferiors.Add(specilist2);

    // теперь независимо от конкретного типа, реализующего интерфес,
    // можно работать с членами интерфейса,
    // реализация которых инкапсулирована конкретными типами
    foreach (I_Inferior employee in manager.ListOfInferiors)
        if (!employee.IsWorking) employee.Work();
}
```

### 9.5. Интерфейсные индексаторы, свойства

Как уже упоминалось в интерфейсе тело методов и свойств не указывается, только сигнатура. Поэтому свойства интерфейса имеют следующий вид:

```
public interface I_Manager
{
    List<I_Inferior> ListOfInferiors { get; }
}
```

Индексатор (indexer) – это специальный вид свойства. Он позволяет обращаться к объектам классов коллекций, являющихся членами класса, с помощью квадратных скобок. Общая форма индексатора объявленного в интерфейсе имеет вид:

```
element_type this [int index]
{
    get;
    set;
}
```

get и set представляют собой индексаторы, предназначенные только для чтения и только для записи соответственно. Например, список служащих можно реализовать следующим образом:

```
class ListOfEmployers
{
    private List<Employee> listOfEmployers = new List<Employee>();
    Employee this[int index]
    {
        get { return listOfEmployers[index]; }
        set { listOfEmployers[index] = value; }
    }
}
```

Принято в качестве индексов использовать целые числа, но допустимо индексировать объекты классов коллекций аргументами другого типа, например строками. Более того, можно указать несколько аргументов, создав тем самым целый массив.

### 9.6. Наследование интерфейсов

Как уже упоминалось, класс может наследовать несколько интерфейсов. Причем синтаксис объявления аналогичен наследованию класса. После имени класса наследника ставится двоеточие и через запятую перечисляются интерфейсы, которые наследует класс. Причем если класс наследует несколько интерфейсов, то в нем должны быть реализованы все методы каждого из интерфейсов. Интерфейсы могут наследовать другие интерфейсы. Также, если класс наследует интерфейс, который в свою очередь тоже наследует интерфейс, то классом должны быть реализованы все методы, объявленные в цепочке наследования.

### 9.7. Проблемы сокрытия имен при наследовании интерфейсов

Иногда нужно объявить класс, который наследует несколько интерфейсов. Может возникнуть ситуация, когда у этих интерфейсов есть методы с одинаковым названием. Например, есть два интерфейса:

```
public interface I_Human
{
    object Work();
}

public interface I_Inferior
{
    object Work();
}
```

Мы хотим создать класс, который наследует оба эти интерфейса. Он должен реализовать все методы, которые есть в каждом интерфейсе.

```
public class Employee: I_Human, I_Inferior
{
    //реализация метода объявленного в I_Human
    object I_Human.Work() {...}

    //реализация метода объявленного в I_Inferior
    object I_Inferior.Work() {...}

}
```

Теперь возникает вопрос: когда мы будем вызывать метод `Work()` для экземпляра класса `Employee`, какой из двух методов будет вызван?

Для того чтобы вызвать нужный метод, необходимо привести тип к нужному интерфейсу. Рассмотрим пример:

```
//Создадим экземпляр класса Employee:
Employee john = new Employee();

I_Human human = john;
human.Work(); //вызываем метод I_Human.Work

I_Inferior inferior = john;
inferior.Work();  //вызываем метод I_Inferior.Work
```

# 10. Анализ стандартных интерфейсов

Рассмотрим интерфейсы, которые реализует класс `Array` (массив). При создании переменной массива, ей будут доступны все свойства и методы следующих интерфейсов.

## 10.1. IEnumerable

Поддерживает перебор всех элементов коллекции.

| Свойства и методы | Описание |
|---|---|
| `GetEnumerator` | Возвращает перечислитель, который осуществляет перебор элементов коллекции. |
| `Cast<TResult> (this IEnumerable source)` | Преобразовывает элементы объекта `IEnumerable` в заданный тип. `TResult` - тип, в который преобразуются элементы объекта `source`. |
| `OfType<TResult> (this IEnumerable source)` | Выполняет фильтрацию элементов объекта `IEnumerable` по заданному типу. `TResult` - тип, по которому фильтруются элементы последовательности. `source` - объект `Ienumerable(T)`, содержащий элементы входной последовательности типа `TResult`. |

## 10.2. IEnumerator

Этот интерфейс использует оператор `foreach` для прохода по всем элементам массива. Он имеет следующие свойства:

| Свойства и методы | Описание |
|---|---|
| `MoveNext()` | Переходит на следующий элемент коллекции. Если такой существует – метод возвращает `true`. Иначе – `false`. |
| `Current` | Свойство, которое возвращает текущий элемент. |
| `Reset()` | Переводит курсор в начало коллекции. |

## 10.3. ICollection

Наследует `IEnumerable`, имеет дополнительные свойства и методы.

| Свойства и методы | Описание |
|---|---|
| `Count` | Свойство, которое возвращает целое число – количество элементов коллекции. |
| `IsSynchronized` | Свойство, которое возвращает значение true, если доступ к классу `ICollection` является синхронизированным (потокобезопасным), в противном случае — `false`. Для массива это |

| | |
|---|---|
| | свойство всегда возвращает – `false`. |
| `CopyTo(Array array, int index)` | Копирует элементы коллекции `ICollection` в массив array, начиная с указанного индекса массива `Array index`. |

### 10.4. IList

Наследует `ICollection`, является базовым интерфейсом для всех неуниверсальных списков. Имеет следующие дополнительные свойства.

| Свойства и методы | Описание |
|---|---|
| `int Add(object value)` | Добавляет в коллекцию элемент `value`. Возвращает номер позиции, в которую был добавлен элемент. |
| `void Clear()` | Данная процедура удаляет все элементы из списка. |
| `bool Contains(object value)` | Возвращает `true`, если элемент `value` содержится в списке. Иначе возвращает `false`. |
| `int IndexOf(object value)` | Возвращает индекс элемента `value` в данном списке. |
| `void Insert(int index, object value)` | Вставляет элемент `value` в список на заданную позицию. |
| `void Remove(object value)` | Удаляет первое вхождение элемента `value` в списке. |
| `void RemoveAt(int index)` | Удаляет из списка элемент, который располагается по указанному индексу `(index)`. |
| `bool IsFixedSize` | Свойство, которое возвращает `true`, если список имеет фиксированный размер. Иначе - `false`. |
| `bool IsReadOnly` | Возвращает `true`, если коллекция доступна только для чтения. Иначе – `false`. |
| `object Item[int index]{get;set;}` | Получает или задает объект по заданному индексу. |

### 10.5. IDisposable

Этот интерфейс является альтернативой деструктору. Определенные в нем методы позволяют освобождать неуправляемые ресурсы.

| Свойства и методы | Описание |
|---|---|
| `void Dispose()` | Выполняет удаление, высвобождение и сбор неуправляемых ресурсов. |

### 10.6. IComparable

Используется для сортировки элементов. Содержит метод, который определяет, как будут сравниваться элементы.

| Свойства и методы | Описание |
|---|---|
| `int CompareTo(object obj)` | Сравнивает текущий экземпляр с другим объектом того же типа и возвращает целое число, которое показывает, расположен ли текущий экземпляр перед, после или на той же позиции в порядке сортировки, что и другой объект. |

## Домашнее задание

**Задание 1.** Для пространства имен `System.Windows.Forms` используя MSDN Library проанализировать следующие классы и их взаимоотношения: `ScrollableControl`, `ContainerControl`, `Form`, `UserControl`, `PropertyGrid`, `SplitContainer`, `ToolStripContainer`, `ToolStripPanel`, `UpDownBase`, `DomainUpDown`, `NumericUpDown`, `Panel`, `ToolStrip`, `Control`, `ButtonBase`, `Button`, `CheckBox`, `RadioButton`, `DataGrid`, `DataGridView`, `DateTimePicker`, `GroupBox`, `Label`, `ListControl`, `ListBox`, `ListView`, `MonthCalendar`, `PictureBox`, `ProgressBar`, `PrintPreviewControl`, `ScrollableControl`, `ScrollBar`, `Splitter`, `StatusBar`, `TabControl`, `TextBoxBase`, `ToolBar`, `TreeView`, `TrackBar`. Построить диаграммы классов.

**Задание 2.** Разработать абстрактный класс ГеометрическаяФигура со свойствами ПлощадьФигуры и ПериметрФигуры. Разработать классы-наследники: Треугольник, Квадрат, Ромб, Прямоугольник, Параллелограмм, Трапеция, Круг, Эллипс и реализовать конструкторы, которые однозначно определяют объекты данных классов.

Реализовать интерфейс ПростойНУгольник, который имеет свойства: Высота, Основание, УголМеждуОснованиемИСмежнойСтороной, КоличествоСторон, ДлиныСторон, Площадь, Периметр.

Реализовать класс СоставнаяФигура который может состоять из любого количества ПростыхНУгольников. Для данного класса определить метод нахождения площади фигуры.

Предусмотреть классы исключения невозможности задания фигуры (введены отрицательные длины сторон; или при создании объекта треугольника существует пара сторон, сума длин которых меньше длины третей стороны; и т.п.)

Создать диаграммы взаимоотношений классов и интерфейсов.

**Задание 3.** Разработать архитектуру классов иерархии товаров при разработке системы управления потоками товаров для дистрибьюторской компании. Прописать члены классов. Создать диаграммы взаимоотношений классов и интерфейсов.

Должны быть предусмотрены разные типы товаров, в том числе:

- бытовая химия,
- продукты питания.

Предусмотреть интерфейсы:

- скоропортящиеся продукты,
- ликероводочные продукты и табачные изделия (акцизные продукты),
- легковоспламеняющиеся товары,
- бьющиеся товары.

Предусмотреть классы управления потоком товаров (пришло, реализовано, списано, передано).

Реализовать собственные классы исключений:

- «нет в наличие»,
- «истек срок годности»,
- «бракованный товар».

## Рекомендуемая литература

1. Шмуллер, Джозеф. Освой самостоятельно UML за 24 часа, 3-е издание.: Пер. с англ. – М.: «Вильямс», 2005. – 416 с.: ил.
2. Фаулер М., Скотт К. UML. Основы. – Пер. с англ. – Спб.: Символ-Плюс, 2002. – 192 с., ил.
3. Дж. Рамбо, М. Блаха. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е. изд. – Сбп.: Питер, 2007. – 544с.: ил.