Н А С Л Е Д О В А Н И Е

НАСЛЕДОВАНИЕ КАК ПРИНЦИП ООП

Наследование — это отношение, связывающее классы, один из которых является ***базовым*** и называется ***родительским***, а другой создается на его основе и называется ***производным***, ***наследником***.

Наследование заключается в том, что класс-наследник приобретает свойства и методы родительского класса и добавляет к ним собственные.

НАСЛЕДОВАНИЕ ПОЗВОЛЯЕТ:

– исключить из программы повторяющиеся фрагменты кода;

– упростить модификацию программы;

– упростить создание новых программ на основе существующих;

– использовать библиотеки классов, когда программист может взять за основу классы, разработанные кем-то другим и создать наследников с требуемыми свойствами.

ТИПЫ НАСЛЕДОВАНИЯ В ООП

- ***Наследование реализации*** (**implementation inheritance**) означает, что производный класс происходит от базового класса, получая от него все поля и методы. Этот тип наследования реализует механизмы расширения базового класса.

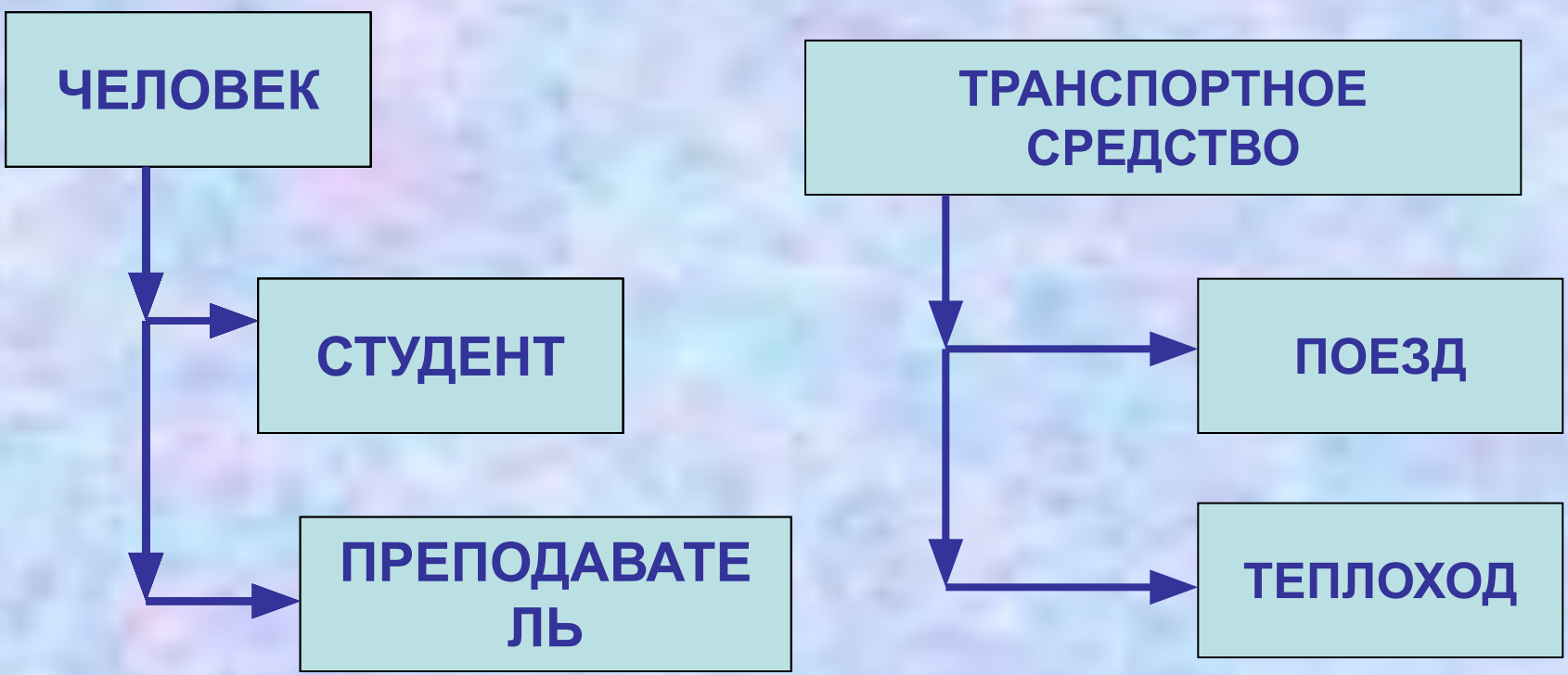
- ***Наследование интерфейса*** (**interface inheritance**) означает, что производный класс наследует только сигнатуру методов базового класса, но не наследует никакой реализации. Наследование интерфейса часто трактуется как выполнение контракта: наследуя интерфейс, производный класс берет на себя обязанность предоставить клиентам определенную функциональность.

НАСЛЕДОВАНИЕ

РЕАЛИЗАЦИИ (отношение is a)

В ЯЗЫКЕ C#

ПРИМЕРЫ НАСЛЕДОВАНИЯ IS - A



ОБЪЯВЛЕНИЕ ПРОИЗВОДНОГО КЛАССА

[спецификатор доступа] [модификатор класса]

class имя_производного_класса : базовый класс

{

тело_класса

}

ФРАГМЕНТ ПРОГРАММЫ НАСЛЕДОВАНИЯ КЛАССОВ

```
class A {       // базовый класс
   int a;    // закрытое поле базового класса
// методы базового класса
   public A ()    // конструктор класса A
     {…}
   public void F()  // метод базового класса
     {…}
          }
class B : A {       // производный класс
   int b;    // закрытое поле производного класса
// методы производного класса
   public B ()    // конструктор класса A
     {…}
   public void G()
     {…}
          }
```

Доступ к наследуемым элементам

```
class A
   {
      int d=5;     // закрытое поле базового класса
   }
   class B : A
   {
      public void Print ()
      {
// метод Print производного класса не имеет доступа к
// закрытому полю d базового класса
// ошибка
         Console.WriteLine("значение унаследованного поля = {0}", d);
      }
     }
```

Важно! **Закрытый элемент класса недоступен для любого кода, определенного вне этого кода, в том числе и для производных классов.**

СПОСОБЫ ОБЕСПЕЧЕНИЯ ДОСТУПА К ЭЛЕМЕНТАМ БАЗОВОГО КЛАССА

1. С помощью спецификаторов доступа **protected** и **public** (не рекомендуется).

2. Через общедоступные методы-свойства базового класса.

ПРИМЕР ДОСТУПА ИЗ ПРОИЗВОДНОГО КЛАССА К protected полю базового класса

```
class A    {
      protected int d=5;  // объявление защищенного поля
   }
   class B : A    {
      public void Print()
      {
// защищенное поле d доступно в производном классе
         Console.WriteLine("значение унаследованного поля = {0}", d);
      }
   }
   class Program {
      static void Main(string[] args)
      {
         A obj = new A();
// ошибка! Класс  Program не является производным от класса A, поэтому  защищенное
 // поле d не доступно методу Main класса  Program,  этот код не будет скомпилирован
         obj.d = 8;
      }
}
```

ДОСТУП К ПОЛЮ БАЗОВОГО КЛАССА ЧЕРЕЗ ЕГО МЕТОДЫ-СВОЙСТВА

```
class A   {  // базовый класс
   int d=5;
public int D   // определение public - свойства D в базовом классе
     {
        get
        {   return d;      }
     }
  }
  class B : A   {  // производный класс
     public void Print ()
     {
// доступ  к закрытому полю d базового  класса A через public - свойство D базового класса A
        Console.WriteLine("значение унаследованного поля = {0}", D);
     }
  }
  class Program {
     static void Main(string[ ] args)       {
        B obj = new B();
        obj.Print();
// Доступ к закрытому полю d класса A через public - свойство D класса A
        Console.WriteLine("значение  поля d в классе A = {0}", obj.D);
     }
    }
```

Конструкторы классов и наследование

1. Базовый и производный классы могут не иметь конструкторов. В этом случае при создании объекта производного класса автоматически вызываются конструкторы, созданные компилятором по умолчанию.

2. Если при создании объекта производного класса необходимо выполнить инициализацию его полей конкретными значениями, то в производном классе необходимо определить конструктор с параметрами. **Базовый класс может иметь конструктор по умолчанию или созданный программистом.**

Определение в производном классе конструктора с параметрами (пример)

```
class A   { // в базовом классе конструктор не определен
    int  d;   // закрытое поле базового класса
     public int D
      {
         get  // свойство чтения значения поля d
         {          return d;    }
        set   // свойство установки значения поля d
        {          d=value;       }
      }
              }
  class B : A   {
      int k;    // закрытое поле производного класса
public  B   (  int d1, int k)   { // конструктор с параметрами производного класса
// инициализация закрытого поля d в базовом классе путем обращения к  методу-свойству D базового класса.
           D = d1;
        this.k = k; // инициализация собственного закрытого поля k
      }
      public void Print()
      {
         Console.WriteLine("значение унаследованного поля = {0}", D);
         Console.WriteLine("значение собственного поля = {0}", k);
      }
   }
```

Определение в производном классе конструктора с параметрами (продолжение примера)

```
class Program
  {
     static void Main(string[] args)
     {
        B obj = new B(3,5);
        obj.Print();
     }
  }
}
```

Примечание. При использовании конструктора базового класса по умолчанию, он вызывается неявно.

ЯВНЫЙ ВЫЗОВ КОНСТРУКТОРА БАЗОВОГО КЛАССА

Синтаксис:

```
имя_производного_класса (список параметров) :
    base (список параметров)
  {
    // тело конструктора
    }
```

Список параметров конструктора производного класса должен состоять из параметров, необходимых для передачи конструктору базового класса и параметров, необходимых для инициализации собственных полей.

Определение в базовом и производном классах конструкторов с параметрами (пример)

```
class A   //базовый класс
  {
   int  d;
   public A(int d)  // конструктор  с параметрами базового класса
   {
      this.d = d;
   }
    public int D
      {
         get
         {      return d;        }
        set
         {         d=value;      }
      }
  }
```

Определение в базовом и производном классах конструкторов с параметрами (продолжение примера)

```
class B : A    {  //производный класс
      int k;

// конструктор  с параметрами производного класса
// выполняет явный вызов конструктора базового класса
     public  B (int d, int k) : base (d)
      {
         this.k = k;
}
      public void Print()
      {
        Console.WriteLine("значение унаследованного поля = {0}", D);
          Console.WriteLine("значение собственного поля = {0}", k);
      }
  }
```

Определение в базовом и производном классах конструкторов с параметрами (продолжение примера)

```
class Program
{
    static void Main(string[] args)
    {
        B obj = new B(3,5);
        obj.Print();
    }
}
```

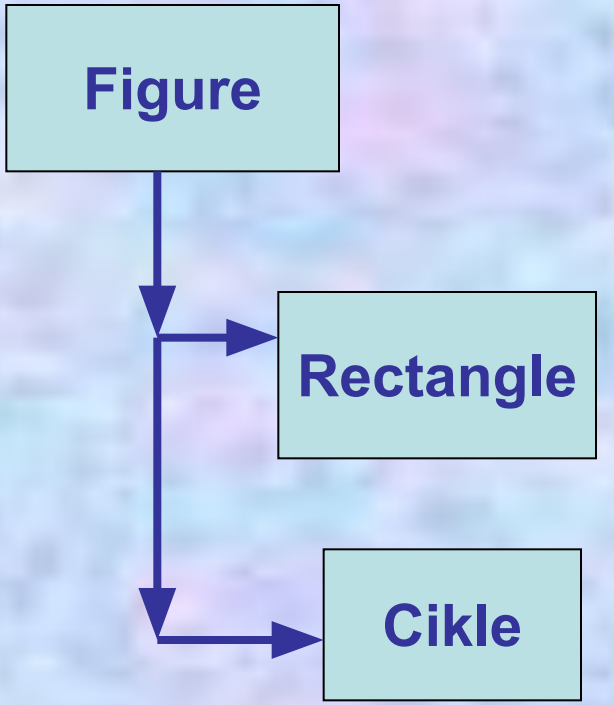
Примечание. Базовый класс может иметь несколько конструкторов, каждый из которых может быть вызван с помощью ключевого слова base. Если в конструкторе производного класса отсутствует служебное слово base, то автоматически вызывается конструктор базового класса, определенный по умолчанию.

Виртуальные методы. Динамическое связывание

ЦЕЛЬ ПРИМЕНЕНИЯ

К механизму **виртуальных функций** (***virtual function***) обращаются в тех случаях, когда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Виртуальные функции важны потому, что они используются для поддержки **динамического полиморфизма**.

ПРИМЕР



Пусть у базового класса **Figure** есть метод **Build** построения этой фигуры. В производных классах **Rectangle** и класс **Circle** мы хотим реализовать свои методы **Build** для построения прямоугольника и окружности соответственно. В таком случае в производных классах следует переопределить метод **Build** из базового класса.

Переопределение метода базового класса в производных (необходимые шаги)

1. В базовом классе заголовок метод должен содержать служебное слово **virtual**.
2. В производных классах переопределяемые методы должны быть записаны со служебным словом **override**.

Переопределение метода базового класса в производных (пример)

```
// базовый класс Figure
      class Figure
      {
// определение виртуального метода Build в базовом
   классе Figure
         public virtual void Build()
         {
Console.WriteLine("Строим геометрическую
   фигуру");
         }
      }
```

Переопределение метода базового класса в производных (продолжение примера)

```
// производный класс Rectangle
      class Rectangle : Figure
      {
// переопределение виртуального метода Build в
 // производном классе Rectangle
          public override void Build()
          {
           Console.WriteLine("Строим прямоугольник");
          }
      }
```

Переопределение метода базового класса в производных (продолжение примера)

```
// производный класс Circle
      class Circle : Figure
      {
// переопределение виртуального метода Build в
 // производном классе Circle
          public override void Build()
          {
              Console.WriteLine("Строим окружность");
          }
      }
```

Переопределение метода базового класса в производных (продолжение примера)

```
static void Main(string[ ] args)      {
// создание массива из трех ссылок на объекты типа Figure
        Figure[ ] obj = new Figure[3];
// инициализация ссылки адресом объекта базового  класса
        obj[0] = new Figure();
// инициализация ссылки адресом объекта производного  класса
        obj[1] = new Rectangle();
// инициализация ссылки адресом объекта производного  класса
        obj[2] = new Circle();
      for (int i = 0; i < obj.Length; i++)
// вызывается виртуальный метод соответствующего класса
          obj[i].Build();
    }
```

Эта программа выводит следующие строки:
Строим геометрическую фигуру
Строим прямоугольник
Строим окружность

Реализация метода динамического связывания

Для виртуальных методов компилятор формирует ***таблицу виртуальных методов*** (**Virtual Method Table, VMT**). Для каждого класса создается одна таблица. Каждый объект во время выполнения программы должен иметь доступ к **VMT**.

Обеспечение этой связи нельзя поручить компилятору, так как она должна устанавливаться во время выполнения программы при создании объекта. Поэтому связь экземпляра объекта с **VMT** устанавливается с помощью специального кода, автоматически помещаемого компилятором в конструктор объект.

Примечание. В связи с тем, что вызов виртуального метода, в отличие от обычного, выполняется через дополнительный этап получения адреса метода из таблицы адресов **VMT**, это несколько замедляет выполнение программы.

АБСТРАКТНЫЕ КЛАССЫ

ЦЕЛЬ ПРИМЕНЕНИЯ

Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактные классы служат только для порождения потомков. Как правило, в абстрактных классах создается набор методов, которые каждый из потомков будет реализовывать по-своему.

Абстрактными называются классы, не допускающие создания объектов.

Абстрактные классы (пример)

```
// Объявляем класс Figure абстрактным, запрещая создание
// объектов этого класса
abstract public class Figure
{
// Открытые интерфейсы и внутренние данные
   класса
}
```

Теперь при попытке создания объекта класса Figure компилятор будет выдавать сообщение об ошибке.

```
// Ошибка! Нельзя создавать экземпляры абстрактного класса
Figure X = new Figure ();
```

Абстрактные методы

Абстрактные методы **– это методы, имеющие пустые тела.**

При объявлении абстрактного метода используется следующая форма синтаксиса:

abstract тип имя (список параметров);

В этом объявлении отсутствует тело метода. Например, в абстрактном классе Figure может быть определен следующий метод:

public abstract void Build ();

Переопределение абстрактного метода базового класса в производных классах (пример)

```
// объявление класса Figure как абстрактного
    abstract class Figure    {
        public abstract void Build();
    }
                class Rectangle : Figure {
// переопределение абстрактного метода Build в классе Rectangle
            public override void Build()
            {
                Console.WriteLine("Строим прямоугольник");
            }
        }
 class Circle : Figure       {
// переопределение абстрактного метода Build в классе Circle
            public override void Build()
            {
                Console.WriteLine("Строим окружность");
            }
        }
```

Переопределение абстрактного метода базового класса в производных классах (продолжение примера)

```
class Program
      {
      static void Main(string[] args)
      {
// в методе Main теперь можно создавать объекты только
 // производных классов
         Rectangle obj1 = new Rectangle();
         obj1.Build();
         Circle obj2 = new Circle();
         obj2.Build();
      }
      }
```

Класс System.Object

БАЗОВЫЙ КЛАСС ДЛЯ ВСЕХ КЛАССОВ

В языке **C#** все типы данных (как структурные, так и ссылочные) производятся от единого общего предка: класса **System.Object**. Класс **System.Object** определяет общее полиморфическое поведение для всех типов данных во вселенной **.NET**.

При объявлении любого класса можно явно указать, что наш класс производится от **System.Object**:

```
class Figure : System.Object
    {
     ..........
..... }
```

Основные методы класса System.Object

Метод	Назначение
Equals()	По умолчанию этот метод возвращает «истинно» только тогда, когда сравниваемые сущности указывают на одну и ту же область в оперативной памяти. Поэтому этот метод в его исходной реализации предназначен только для сравнения объектов ссылочных типов, но не структурных. Однако помните, что если вы замещаете этот метод, вам потребуется также заместить метод **GetHashCode()**
GetHashCode()	Возвращает целочисленное значение, идентифицирующее конкретный экземпляр объекта данного типа.
GetType()	Метод возвращает объект **Type()**, полностью описывающий тот объект, из которого метод был вызван. Это метод идентификации времени выполнения (**Runtime Type Identification, RTTI**), который предусмотрен во всех объектах
ToString()	Возвращает символьное представление объекта в формате <имя пространства имен>.<имя класса> (такой формат носит также название «полностью определенного имени» — **fully qualified name**).
MemberwiseClone();	Этот метод предназначен для создания еще одной ссылки на область, занимаемую объектом данного типа в оперативной памяти. Этот метод не может быть замещен. Если вам потребовалось реализовать поддержку создания полной копии объекта в оперативной памяти, вы должны реализовать а вашем классе поддержку интерфейса **ICloneable**.

Фрагмент определения класса System.Object

- namespace System
- {
- public class Object
- {
- public Object();
- public virtual Boolean Equals (Object obj);
- public virtual Int32 GetHashCode();
- public Type GetType();
- public virtual String ToString();
- protected virtual void Finalize();
- projected Object MemberwiseClone();
- }
- ……
- }

Демонстрация методов, унаследованных от класса System.Object

```
class ObjTest    {
      static void Main(string[ ] args)       {
         ObjTest c1 = new ObjTest();
Console.WriteLine("ToString: {0}", c1.ToString());   // Выводим информацию на консоль
Console.WriteLine("Hash Code: {0}", c1.GetHashCode());
Console.WriteLine("Type: {0}", c1.GetType().ToString());
ObjTest c2 =c1;  // Создаем еще одну ссылку на c1
object o = c2;
// Действительно ли все три экземпляра указывают на одну  область в оперативной памяти?
if (o.Equals(c1) && c2.Equals(o))
Console.WriteLine(" Ссылки установлены на одну область памяти!");
     }
  }
```

Эта программа выводит следующие строки

```
ToString: ConsoleApplication1.ObjTest
Hash Code: 58225482
Type: ConsoleApplication1.ObjTest
 Ссылки установлены на одну область памяти!
Для продолжения нажмите любую клавишу . . .
```

Замещение методов класса System.Object (пример)

```
class Person
{
// Данные о человеке
    public string FirstName;   //Фамилия
    public string LastName;   //Имя
    public string SSN;      // номер социального страхования
    public byte age;         //возраст
// конструктор с параметрами
    public Person(string fname, string lname, string ssn, byte a)
    {
        FirstName = fname;
        LastName = lname;
        SSN = ssn;
        age = a;
    }
```

Замещение методов класса System.Object (продолжение примера)

```
// замещаем метод Equals(), унаследованный от System.Object
// метод  возвращает истинно тогда, когда у сравниваемых объектов типа
// Person  внутренние состояния одинаковы (то есть значения полей
// firstName,  lastName, SSN и age совпадают)
        public override bool Equals(object o)
        {
// выполняется сравнение значений переменных объекта,
// принимаемого в качестве  параметра, со значениями переменных объекта,
// для которого метод Equals был вызван (с помощью ключевого слова this)

            Person temp = (Person) o;
    if (temp.FirstName == this.FirstName &&  temp.LastName
==this.LastName
    && temp.SSN == this.SSN  &&  temp.age == this.age)
            {      return true;            }
            else
       return false;
          }
```

Замещение методов класса System.Object (продолжение примера)

```
// замещаем метод ToString()  так, чтобы  он
// возвращал не имя типа, а информацию о внутреннем состоянии  объекта
// возвращаемое значение типа string заключено в прямоугольные скобки [ ]
        public override string ToString()
        {
     StringBuilder sb = new StringBuilder();
    sb.AppendFormat("[FirstName= {0}", this.FirstName);
    sb.AppendFormat(" LastName= {0}", this.LastName);
    sb.AppendFormat(" SSN= {0}", this.SSN);
     sb.AppendFormat(" Age= {0}]", this.age);
            return sb.ToString();
        }
```

Замещение методов класса System.Object (продолжение примера)

```
// замещение метода GetHashCode()
        public override int GetHashCode()
        {
// Возвращаем хэш-код, основанный на номере социального
// страхования  (переменной SSN)
            return SSN.GetHashCode();
        }
    }
```

Примечание. **При замещении метода EqualsO следует также заместить метод GetHashCode (). В случае, если это сделано не будет, компилятор выдаст предупреждение. Метод GetHashCode() возвращает числовое значение, идентифицирующее объект в оперативной памяти. Чаще всего это значение используется в коллекциях, работающих с хэш-кодами объектов**.

Замещение методов класса System.Object (продолжение примера)

```
 static void Main(string[ ] args)       {
          Console.WriteLine("***** Тестирование класса  Person*****");
          Person p1 = new Person("Иванов", "Иван", "222-22-2222", 25);
          Person p2 = new Person("Иванов", "Иван", "222-22-2222", 25);
// используем замещенный метод Equals для сравнения состояния  объектов p1 и p2
            if (p1.Equals(p2) && p1.GetHashCode() == p2.GetHashCode())
              Console.WriteLine("Объекты P1 и P2 одинаковы\n");
          else       Console.WriteLine("Объекты P1 и P2 различны\n");
p2.age = 2; // меняем внутренне состояние объекта  p2
// еще раз производим сравнение внутреннего состояния объектов p1 и p2
            if (p1.Equals(p2) && p1.GetHashCode() == p2.GetHashCode())
              Console.WriteLine("Объекты P1 и P2 равны\n");
          else  Console.WriteLine("Объекты P1 и P2 различны\n");
Console.WriteLine(p1.ToString());
          Console.WriteLine(p2);
      }
```

Программа выводит следующие строки:

```
***** Тестирование класса  Person*****
Объекты P1 и P2 одинаковы
Объекты P1 и P2 различны
[FirstName= Иванов LastName= Иван SSN= 222-22-2222 Age= 25]
[FirstName= Иванов LastName= Иван SSN= 222-22-2222 Age= 2]
```

ЗАДАНИЯ НА ЛАБОРАТОРНУЮ РАБОТУ

При выполнении заданий требуется написать законченную программу, в которой реализуется наследование классов.

В программе требуется описать базовый и производные классы. Базовый класс (возможно, абстрактный) с помощью виртуальных или абстрактных методов и свойств должен задавать интерфейс для производных классов. Во всех классах следует переопределить метод **Equals**, чтобы обеспечить сравнение значений, а не ссылок. Функция **Main** должна содержать массив из элементов базового класса, заполненный ссылками на производные классы. В этой функции должно демонстрироваться использование всех разработанных элементов классов.

ВАРИАНТЫ ЗАДАНИЙ

- 1. Студент, преподаватель, персона, заведующий кафедрой.
- 2. Служащий, персона, рабочий, инженер.
- 3. Рабочий, кадры, инженер, администрация.
- 4. Организация, страховая компания, завод.
- 5. Журнал, книга, печатное издание, учебник.
- 6. Тест, экзамен, выпускной экзамен, испытание.
- 7. Игрушка, продукт, товар, молочный продукт.
- 8. Квитанция, накладная, документ, счет.
- 9. Автомобиль, поезд, транспортное средство, экспресс.
- 10. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
- 11. Республика, монархия, королевство, государство.
- 12. Млекопитающее, парнокопытное, животное, птица.

ВАРИАНТЫ ЗАДАНИЙ

- 13. Корабль, пароход, парусник, корвет
- 14. Создать абстрактный класс Vehicle (транспортное средство). На его основе реализовать классы Plane (самолет), Car (автомобиль) и Ship (корабль). Классы должны иметь возможность задавать и получать координаты и параметры средств передвижения (цена, скорость, год выпуска и т. п.) с помощью свойств. Для самолета должна быть определена высота, для самолета и корабля количество пассажиров, для корабля — порт приписки. Динамические характеристики задать с помощью методов.
- 15. Автомобиль, грузовик, автобус, легковой автомобиль.
- 16. Помещение, учебная аудитория, кабинет, спортзал.
- 17. Многоугольник, прямоугольник, квадрат, треугольник.
- 18. Вычислительная сеть, локальная сеть, глобальная сеть.
- 19. Программный продукт, архиватор, среда программирования, текстовый редактор.
- 20. Электронная вычислительная машина, персональный компьютер, ноутбук.

ИНТЕРФЕЙСЫ

ОПРЕДЕЛЕНИЯ

В ООП

Интерфейс — множество операций, которое определяет набор услуг (службу), предоставляемых классом или компонентом

В ЯЗЫКЕ C#

Интерфейсом **называется чисто абстрактный класс, содержащий только описания без реализации**

ЦЕЛИ ПРИМЕНЕНИЯ

1. Определение действий, выполняемых классом, без указания способа их выполнения, т.е. необходимость отделить описание класса от реализации.

2. Поддержка множественного наследования в языке C#

СИНТАКСИС ИНТЕРФЕЙСА

```
[атрибуты][спецификаторы] interface
имя_интерфейса [:список_родителей]
{
тело_класса
}
```

Спецификаторы:

new, public, protected, internal и private.

Спецификатор new применяется для вложенных интерфейсов. Остальные спецификаторы управляют видимостью интерфейса. По умолчанию интерфейс доступен только из сборки, в которой он описан(internal).

ПРИМЕР ОБЪЯВЛЕНИЯ

```
// Этот интерфейс определяет возможность
 // нарисовать геометрическую фигуру

interface IDraw
{
// автоматически (неявный образом) этот член
// интерфейса становится абстрактным

void Draw();
}
```

***Примечание*. Имена интерфейсов принято начинать с буквы I.**

ОТЛИЧИЯ ИНТЕРФЕЙСА ОТ АБСТРАКТНОГО КЛАССА

1. При объявлении интерфейса все его методы неявно имеют модификатор **public**, и явное указание модификатора доступа приведет к ошибке (даже если он будет **public**);
2. методы интерфейса не содержат тела с реализацией. После объявления метода следует точка с запятой;
3. интерфейс не может содержать данных, в нем могут быть объявлены только методы, события, свойства и индексаторы;
4. класс, наследующий интерфейс, обязан полностью реализовать все методы интерфейса, в то время как потомок абстрактного класса может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом;
5. класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

Реализация интерфейса

После объявления интерфейса следует создать класс (или классы), реализующий интерфейс. Класс, исполняющий интерфейс, называется *интерфейсным*. При объявлении интерфейсного класса сначала указывается базовый класс, если он есть, а затем имя интерфейса:

```
class  имя класса : имя интерфейса
     {
 тело класса
      }
```

Тело интерфейсного класса может содержать не только реализацию методов интерфейса, но и не описанные в интерфейсе методы, а также поля, свойства и события. Однако он *должен* содержать реализацию всех интерфейсных методов.

Пример реализации интерфейса IDraw в классах Rect и Ellipse

```
interface IDraw   // объявление интерфейса
   {
      void Draw();
   }
class Rect : IDraw  {   // интерфейсный класс Rect
// переопределение интерфейсного метода Draw() в классе Ellipse
      public void Draw()
      {
         Console.WriteLine("Рисуем прямоугольник");
      }
   }
class Ellipse : IDraw  {  // интерфейсный класс Ellipse
// переопределение интерфейсного метода Draw() в классе Ellipse
      public void Draw()
      {
         Console.WriteLine("Рисуем эллипс");
      }
   }
```

Пример реализации интерфейса IDraw в классах Rect и Ellipse (продолжение)

```
 static void Main(string[ ] args) {
// объявление переменной val как ссылки на интерфейс IDraw. Эта ссылка
// может ссылаться на любой объект, который реализует интерфейс IDraw
        IDraw[ ] val = new IDraw[2];
// ссылка на интерфейс получает адрес объекта интерфейсного класса Rect
        val[0] = new Rect();
// ссылка на интерфейс получает адрес объекта интерфейсного класса Ellipse
        val[1] = new Ellipse();
        for (int i = 0; i < val.Length; i++)
         {
            val[i].Draw();
         }
    }
```

Результаты работы этой программы имеют вид:
Рисуем прямоугольник
Рисуем эллипс

НАСЛЕДОВАНИЕ

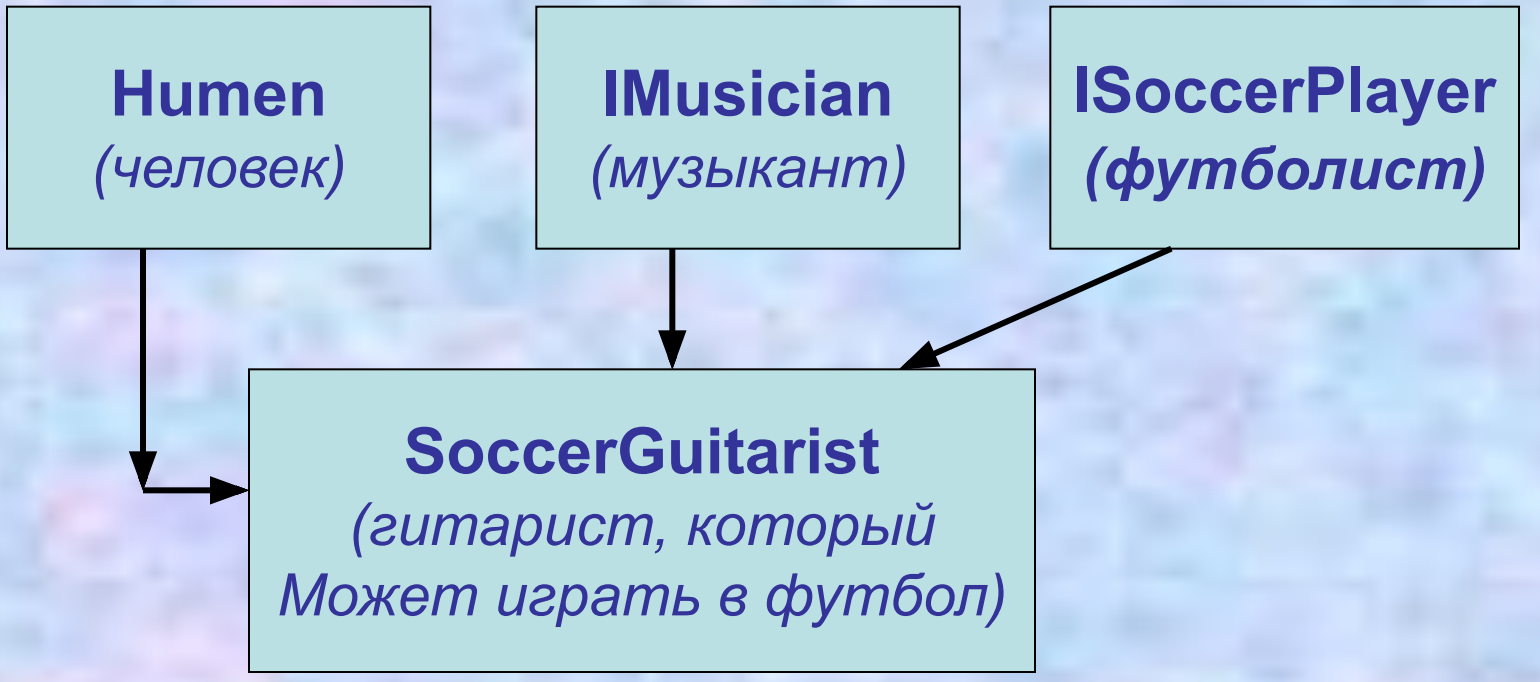
ИНТЕРФЕЙСОВ

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Один класс может реализовывать (или наследовать) несколько интерфейсов одновременно. Это значит, что производный класс в языке C# может наследоваться только от одного базового класса и любого количества интерфейсов.

Каждый класс в языке C# имеет один базовый класс System.Object и дополнительно может иметь любое количество базовых интерфейсов.

Наследование от нескольких базовых интерфейсов



Наследование от нескольких базовых интерфейсов (пример)

```
interface IMusician    {    // интерфейс «музыкант»
        void Tune ();
        void Play();
      }
interface ISoccerPlayer    {  // интерфейс «футболист»
        void Play();
    }
class Human    {          //  базовый клас «человек»
        public Human()
        {
            Console.WriteLine("Я человек");
                  }
    }
```

Наследование от нескольких базовых интерфейсов (продолжение примера)

```
// класс SoccerGuitarist производный от  Human и реализует  интерфейсы классов
// IMusician и ISoccerPlayer

  class SoccerGuitarist : Human, IMusician, ISoccerPlayer {
      public void Tune()
      {
         Console.WriteLine("Я гитарист");
      }

// IMusician.Play()  – это явная реализация метода  Play()  в классе  SoccerGuitarist, требует
//  указания полного  имени метода, которое  включает имя его интерфейса – IMusician
// модификатор доступа при явной реализации указывать нельзя!
    void IMusician.Play()  {   Console.WriteLine("Я играю на гитаре");        }

// ISoccerPlayer.Play()  – это явная реализация метода  Play() в классе SoccerGuitarist,требует
// указания полного имени метода, которое включает имя его интерфейса – ISoccerPlayer
  void ISoccerPlayer.Play()
      {
         Console.WriteLine("Я играю в футбол");
      }
  }
```

Наследование от нескольких базовых интерфейсов (продолжение примера)

```
static void Main(string[] args)
        {
// при создании объекта производного класса неявно вызывается конструктор
// по умолчанию базового класса Human
            SoccerGuitarist val = new SoccerGuitarist();
             val.Tune();
// при доступе к методу Play() выполняется явное приведение ссылки val
// к типу интерфейса, метод которого мы вызываем
            ((IMusician)val).Play();
            ((ISoccerPlayer) val).Play();
        }
```

Результаты работы этой программы имеют вид:

Я человек
Я гитарист
Я играю на гитаре
Я играю в футбол

Особенности реализации методов явным образом при наследовании от нескольких интерфейсов:

1. При доступе к методу необходимо явно приводить ссылку к типу интерфейса, метод которого мы вызываем, или вызвать метод через ссылку на этот интерфейс.

2. Нельзя указывать для реализуемого метода модификатор доступа.

3. Нельзя объявить метод как виртуальный.

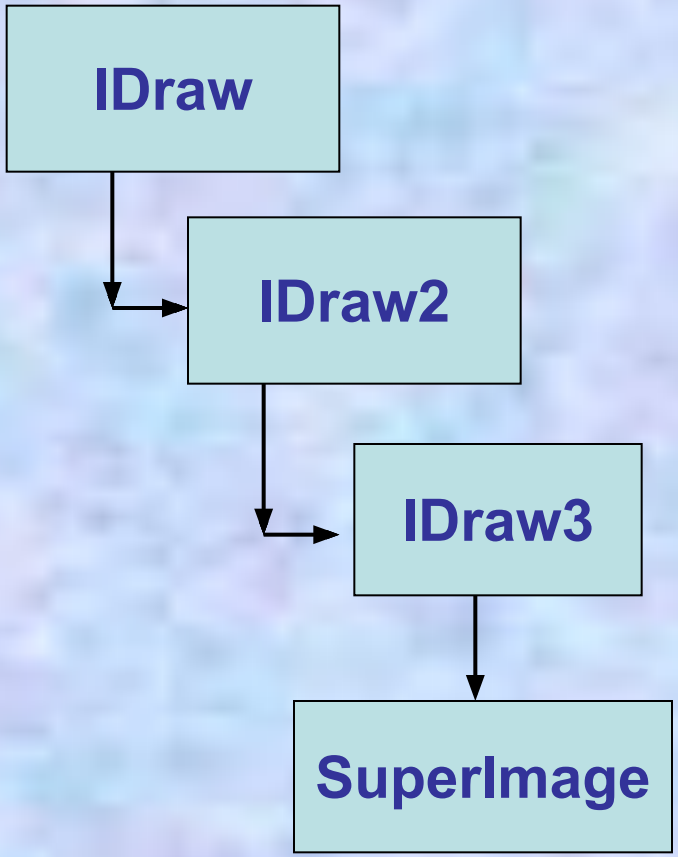
Создание иерархий интерфейсов

```
// Базовый интерфейс IDraw
   interface IDraw    {
      void Draw();
   }

// интерфейс, производный от IDraw
interface IDraw2 : IDraw {
void DrawToPrinter();  }

// интерфейс, производный от IDraw2
interface IDraw3 : IDraw2  {
void DrawToMetaFile();  }

 public class SuperImage : IDraw3
{ }
```



Создание иерархий интерфейсов (пример)

```
public class SuperImage : IDraw3
  {
     // явная реализация  метода интерфейса
     void IDraw.Draw()
     {
        Console.WriteLine("Вывод на консоль...");      // Вывод на консоль
     }
     void IDraw2.DrawToPrinter()
     {
        Console.WriteLine("Вывод на принтер...");      // Вывод на принтер
     }
     void IDraw3.DrawToMetaFile()
     {
        Console.WriteLine("Вывод в файл...");     // Вывод в файл
     }
  }
```

Создание иерархий интерфейсов (продолжение примера)

```
static void Main(string[] args)
      {
         Console.WriteLine("***** Работа с классом SuperImage *****");
         SuperImage si = new SuperImage(); //создание объекта класса
         IDraw itfDraw = (IDraw) si; // Получаем ссылку на интерфейс IDraw
         itfDraw.Draw();
 // получаем ссылку на интерфейс IDraw3, для чего используем оператор is
// оператор is выполняет проверку принадлежности объекта определенному типу
// тип объекта сравнивается с тем, который записан справа от оператора is
 // если объект не поддерживает интерфейс условие станет равно false
         if (itfDraw is IDraw3)          {
            IDraw3 itfDraw3 = (IDraw3)itfDraw;
            itfDraw3.DrawToMetaFile();
            itfDraw3.DrawToPrinter();
         }
```

Результаты работы этой программы имеют вид:

***** Работа с классом SuperImage *****

Вывод на консоль…

Вывод в файл…

Вывод на принтер…

Стандартные интерфейсы .NET. Интерфейс IComparable

Интерфейс IComparable позволяет производить сортировку объектов, основываясь на специально определенном внутреннем ключе.

Определение интерфейса:

```
interface IComparable
  {
  int CompareTo(object o):
   }
```

Интерфейс IComparable (пример)

Пусть пользователь создал массив объектов класса **Car** следующим образом:

```
Car[] myAutos = new Car[5];
myAutos[0] = new Car(123, "Волга");
myAutos[1] = new Car(6, "Ауди");
myAutos[2] = new Car(16, "Мерседес");
myAutos[3] = new Car(13, "Запорожец");
myAutos[4] = new Car(26, "Нива");
```

Задача: Выполнить сортировку созданного массива по различным критериям: номеру автомобиля и его названию.

Интерфейс IComparable (пример)

Для выполнения сортировки по номеру автомобиля в классе Car необходимо выполнить явную реализцию метода **CompareTo интерфейса IComparable:**

```
int IComparable.CompareTo(object o)
    {
        Car temp = (Car)o;
        if(this.CarID > temp.CarID)
            return 1;
        if(this.CarID < temp.CarID)
            return -1;
        else    return 0;
    }
```

Интерфейс IComparer

Для реализации сортировки еще по одному критерию – имени автомобиля воспользуемся возможностями еще одного стандартного интерфейса − **IComparer**, определенного в пространстве имен **System.Collections.**

```
// Стандартный способ сравнения двух объектов
interface IComparer
{
int Compare(object o1, object o2);
}
```

IComparer обычно не реализуется напрямую внутри класса, объекты которого необходимо сортировать (в нашем случае — Car). Этот интерфейс реализуется во вспомогательных классах, которых может быть любое количество — по одному вспомогательному классу на каждую переменную, по которой производится сортировка.

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (пример применения)

```
public class Car : IComparable
    {// этот вспомогательный класс нужен для сортировки объектов
    // полю по PetName   он реализует интерфейс IComparer
    private class SortByPetNameHelper : IComparer
     {
     public SortByPetNameHelper()  {} ;
     // сравниваем имена объектов (PetName)
     int IComparer.Compare(object o1, object o2)       {
           Car t1 = (Car)o1;
           Car t2 = (Car)o2;
           return String.Compare(t1.PetName, t2.PetName);
     }
     }
```

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (продолжение примера применения)

```
// поля класа .
 private int CarID;  //номер автомобиля
 private string petName;   // название автомобиля
 // методы-свойства класса
 public int ID {
      get { return CarID; }
      set { CarID = value; }
 }
 public string PetName     {
      get { return petName; }
      set { petName = value; }
 }
```

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (продолжение примера применения)

```
// конструкторы
  public Car() {} ;
  public Car(int id, string name)
  { this.CarID = id; this.petName = name;}
  //реализация интерфейса  IComparable
  int IComparable.CompareTo(object o)
        {
        Car temp = (Car)o;
        if(this.CarID > temp.CarID)
              return 1;
        if(this.CarID < temp.CarID)
              return -1;
        else return 0;
  }
// свойство возвращает результат сравнения имен автомобилей
  public static IComparer SortByPetName
  { get { return (IComparer) new SortByPetNameHelper(); } }
}
```

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (продолжение примера применения)

```
static void Main(string[] args)      {
            // Создание массива объектов типа  Car
              Car[] myAutos = new Car[5];
              myAutos[0] = new Car(123, "Волга");
              myAutos[1] = new Car(6, "Ауди");
              myAutos[2] = new Car(16, "Мерседес");
              myAutos[3] = new Car(13, "Запорожец");
              myAutos[4] = new Car(26, "Нива");
 // Выводим созданный массив объектов.
  Console.WriteLine("***** Вывод  не отсортированных данных*****");
        foreach(Car c in myAutos)
```

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (продолжение примера применения)

```
    Console.WriteLine("{0} {1}", c.ID, c.PetName);
    // Используем возможности реализованного интерфейса IComparable.
                Array.Sort(myAutos);
    // Выводим информацию из упорядоченного массива
    Console.WriteLine("\n***** Сортировка по номерам автомобилей *****");
          foreach(Car c in myAutos)
                Console.WriteLine("{0} {1}", c.ID, c.PetName);
    // сортировка по названию автомобилей  используем вариант перегруженного
  // в классе System.Array  метода Sort
    // Array.Sort(myAutos, new SortByPetName());
          Array.Sort(myAutos, Car.SortByPetName);
          // Выводим информацию из упорядоченного массива
    Console.WriteLine("\n***** Сортировка по названиям автомобилей *****");
          foreach(Car c in myAutos)
                Console.WriteLine("{0} {1}", c.ID, c.PetName);
      }
```

Стандартные интерфейсы .NET. Интерфейсы IComparable и IComparer (продолжение примера применения)

Результаты работы этой программы имеют вид:

```
***** Вывод  не отсортированных данных*****
123 Волга
6 Ауди
16 Мерседес
13 Запорожец
26 Нива
***** Сортировка по номерам автомобилей *****
6 Ауди
13 Запорожец
16 Мерседес
26 Нива
123 Волга
***** Сортировка по названиям автомобилей *****
6 Ауди
123 Волга
13 Запорожец
16 Мерседес
26 Нива
```

Задание на лабораторную работу №7

Реализуйте для иерархии классов, созданной вами при выполнении лабораторной работы №6 механизм интерфейсов. Используйте стандартные интерфейсы **IComparable** и **IComparer** для выполнения сортировки объектов по различным полям.