НАСЛЕДОВАНИЕ В C#

Наследование – это свойство объектно-ориентированной системы наследовать данные и функциональность базового класса. Можно в класс-потомок к методам и полям родительского класса добавить необходимые поля и методы. Класс-потомок может замещать методы родительского класса. Надо помнить, при изменении родительского класса, класс-потомок может оказаться не рабочим.
Наследование от класса называется расширением базового класса. Если класс A наследует от класса B, то класс A называется потомком, а B – предком. Синтаксически это пишется следующим образом:

```
class A:B {…}
```

Класс-потомок наследует все элементы базового класса, кроме конструктора и деструктора. Все public элементы базового класса остаются неявно public в потомке, private элементы, хоть и наследуются, но доступны только для объектов базового класса. Класс-потомок не может быть более доступным, чем базовый класс.

```
class Example
{
private class NestedBase { }
public class NestedDerived: NestedBase { } // Ошибка
}
```

Класс-потомок имеет доступ ко всем protected полям и методам родительского класса, класс не являющийся потомком доступа к protected членам не имеет.

```
class Token
{
protected string name;
}
class CommentToken:Token
{
public string Name()
{
return name; // Доступ разрешен
}
}

class CommentToken: Token
{
void Fails(Token t)
{
Console.WriteLine(t.name); // Ошибка при компиляции
}
}
```

Для вызова конструктора базового класса из класса-потомка используется ключевое слово base. Вызов конструктора базового класса пишется после заголовка конструктора класса-потомка через двоеточие:

```
A(список параметров): base(параметры для передачи в конструктор базового класса)
{
```

```
//тело конструктора у класса потомка
}
```
Ключевое слово base обращается к базовому классу. Для конструктора по умолчанию вызов базового конструктора производится по умолчанию, то есть его можно не прописывать. Для конструкторов не по умолчанию необходимо явно вызывать конструктор базового класса.

```
class Token
{
protected Token(string name) { ... }
}
class CommentToken: Token
{
public CommentToken(string name) { ... } //ошибка при компиляции
}
```

В примере выше получим ошибку при компиляции, так как будет вызываться конструктор по умолчанию, но он отсутствует в классе Token. Для правильной работы необходимо записать конструктор в следующем виде:
public CommentToken(string name) : base(name) { ... }
Если конструктор базового класса private, то нельзя создавать конструктор класса потомка.

```
class NonDerivable
{
private NonDerivable(){ }
}
class Impossible: NonDerivable
{
public Impossible() { } // Ошибка при компиляции
}
```

В классе-потомке можно переопределять методы базового класса, если они для этого предназначены. Виртуальные методы можно полиморфно переопределять в классах-потомках. В C# по тому, содержит ли базовый класс виртуальные методы, можно определить, был ли этот класс разработан для наследования. Не виртуальный метод имеет только одно определение, одинаковое для всех потомков. Для объявления виртуального метода используется ключевое слово virtual. Виртуальный метод должен содержать тело в базовом классе. Нельзя объявлять виртуальными статичные и private методы. Статичные методы не могут быть виртуальными, так как полиморфизм – это свойство, относящееся к объектам, а не к классам.

Перегруженные методы могут создаваться только для виртуальных методов. Для задания перегруженных методов используется ключевое слово override. В базовом классе метод объявляется виртуальным virtual, в базовом классе есть тело у этого метода, в классе-потомке этот метод определяется перегруженным override и содержит свое тело – таким образом задается перегруженный метод.

```
class Token
{
public virtual string Name() {…}
}
class CommentToken : Token
```

```
{
public override string Name() {…}
}
```

Можно перегружать только абсолютно идентичные методы. Должны совпадать: имя метода, тип возвращаемого значения, список параметров, уровень доступа. Перегруженный метод должен быть virtual или override. Нельзя объявлять перегруженный метод одновременно виртуальным, т.е. override virtual – нельзя. Нельзя, чтобы перегруженные методы были статичными или private.

Можно скрыть наследуемый метод в иерархии классов, заменив его новым идентичным методом при помощи ключевого слова new. Метод

родительского класса не будет наследоваться потомком, и заменится на новый идентичный метод.

```
class Token
{
public int LineNumber() {…}
}
class CommentToken : Token
{
new public int LineNumber() {…}
}
```

Ключевое слово new прячет как виртуальные, так и невиртуальные методы, разрешает проблему совпадения имен, скрывает методы с одинаковой сигнатурой.
В C# можно объявить класс ненаследуемым, при помощи ключевого слова sealed.
public sealed class String {..}
public class MyStr:String; //ошибка – от класса String наследовать нельзя

Использование интерфейсов

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.

```
interface IToken
{
public int LineNumber( ){ … }; // Ошибка при компиляции: 1. Модификатор доступа у метода public //2. Есть тело метода
}
```

C# позволяет наследовать от одного класса и множества интерфейсов. Интерфейс может наследовать от многих интерфейсов.

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
```

```
class Token: IVisitableToken { ... }
Класс может быть более доступным, чем интерфейс
class Example
{
private interface INested { }
public class Nested: INested { } // Разрешено
}
```

Класс должен определить все методы всех интерфейсов, от которых он наследует как напрямую, так и косвенно. Метод интерфейса, определяемый классом должен быть идентичен, то есть должны совпадать параметр доступа, имя, возвращаемое значение и список параметров. Интерфейсные методы, реализуемые в классе, могут быть объявлены как virtual. В этом случае классы наследники могут перегружать эти методы в дальнейшем.

Другой способ реализации интерфейсных методов – явная реализация. При явной реализации необходимо указать полное имя метода: Имя_интерфейса.имя_метода. При явном определении метод не может быть виртуальным, должен отсутствовать модификатор доступа. При вызове метода к нему нет прямого доступа, только через интерфейс.

```
class Token: IToken, IVisitable
{
string IToken.Name( )
{
...
}
private void Example( )
{
Name( ); // Ошибка при компиляции
((IToken)this).Name( ); // Правильно
}
...
}
```

Явная реализация позволяет:

· исключить определение интерфейса из класса, если он не интересен пользователям класса.

· обеспечивать классу несколько определений различных методов интерфейсов одинаковой сигнатуры.

```
interface IArtist
{
void Draw();
}
interface ICowboy
{
void Draw();
}
class ArtisticCowboy: IArtist, ICowboy
{
void IArtist.Draw() {…}
```

```
void ICowbowt.Draw() {…}
}
```

Использование абстрактных классов

Абстрактные классы используются для частичной реализации классов, которые могут быть полностью реализованы в конкретных классах-потомках. Абстрактный класс объявляется с помощью ключевого слова abstract. Правила создания абстрактного класса совпадают с правилами создания обычных классов. Однако, в абстрактных классах можно объявлять абстрактные методы. Нельзя создавать объекты абстрактного класса. Абстрактный класс может являться наследником неабстрактного класса. Все методы интерфейса, определяемого абстрактным классом, должны быть определены в абстрактном классе.

И абстрактные классы, и интерфейсы предназначены для наследования. Однако класс может наследовать только от одного абстрактного класса. Только абстрактные классы могут иметь абстрактные методы. У абстрактного метода отсутствует тело метода. Абстрактные методы – виртуальные, переопределенные абстрактные методы у классов-потомков будут override. Абстрактные методы могут переопределять virtual и override методы.

```
class Token
{
public virtual string Name( ) { ... }
abstract class Force: Token
{
public abstract override string Name( );
}
```

Пример разработки кода

Задача: *Создать базовый класс “Транспорт”. От него наследовать “Авто”, “Самолет”, “Поезд”. От класса “Авто” наследовать классы “Легковое авто”, “Грузовое авто”. От класса “Самолет” наследовать классы “Грузовой самолет” и “Пассажирский самолет”. Придумать поля для базового класса, а также добавить поля в дочерние классы, которые будут конкретно характеризовать объекты дочерних классов. Определить конструкторы, методы для заполнения полей классов (или использовать свойства). Написать метод, который выводит информацию о данном виде транспорта и его характеристиках. Использовать виртуальные методы.*
И так, вы прочитали задачу. Первое, что я рекомендую сделать, это нарисовать, где вам удобно, схему проекта. Классы, поля, методы, возможно интерфейсы и т.д. В общем говоря составьте UML таблицу. Поздравляю, вы уже готовы создавать.
1) Создадим класс *“Транспорт”*. Должно получится следующее*:*

```
namespace ConsoleApp1
{
   public abstract class Transport
      {
      }
}
```

Если вы пишете код в VS у вас будут подключены библиотеки:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Теперь давайте создадим поля, методы, конструктор по умолчанию и с параметрами.

```
namespace ConsoleApplication1
{
   public abstract class Transport
      {
         public int Year { get; set; }
         public int Weight { get; set; }
         public string Color { get; set; }

         protected Transport() { }

         protected Transport(int year, int weight, string color)
         {
            Year = year;
            Weight = weight;
            Color = color;
         }

         public abstract void Info();
      }
}
```

Ура! Мы написали базовый класс от которого будем наследовать дочерние классы. Условно строить машину, класс **Transport** это указания для ВСЕГО транспорта какой год выпуска (поле *Year),* вес *(*поле *Weight*), цвет *(*поле *Color).* И абстрактный метод Info, который будет выводить информацию например так: Машина -Ford Explorer. Вес - 1670 кг. Год - 2019. Цвет - черный и т.д. Еще мы описали 2 типа конструктора:

```
protected Transport() { }
protected Transport(int year, int weight, string color)
{
  Year = year;
  Weight = weight;
  Color = color;
}
```

Конструктор это специальный блок инструкций, вызываемый при создании объекта. То есть, первый инструктор когда мы например создаем объект класса:

```
Transport transport = new Transport();
```

В таком случае мы создадим объект *transport* класса *Transport.* С параметрами по умолчанию. Что это означает? Это означает что поля Year, Weight, Color получат значения

(Year = null, Weight = null, Color = null). Это сделано для того, что бы при выделении памяти в них не было мусора. Также мы можем сделать следующее:

```
protected Transport()
{
            Year = 0;
  Weight = 0;
  Color = "Unknown";
}
protected Transport(int year, int weight, string color)
{
  Year = year;
  Weight = weight;
  Color = color;
}
```

Тут мы явно присвоили полям какие-то свои значения.
Второй конструктор это то же самое присвоение значений, но только когда мы передаем в конструктор *int year, int weight, string color:*

```
Transport transport = new Transport(2019, 1634, "Black");
```

Что такое *protected* и *public?* **Public** — доступ открыт всем другим классам, кто видит определение данного класса. **Protected** — доступ открыт классам, производным от данного. То есть, производные классы получают свободный доступ к таким свойствам или метода. Все другие классы такого доступа не имеют.
Но, так как мы создали не просто класс, а абстрактный класс, нам не удастся создать его объект. Так как объект абстрактного класса создать нельзя.
2) Давайте создадим классы *“Авто”, “Самолет”, “Поезд”:*

```
using System;
namespace ConsoleApp1
{
   class Train : Transport
   {
      public int Carriages { get; set; }
      public Train(int year, int weight, string color, int carriages) : base(year, weight, color)
      {
         Carriages = carriages;
      }

      public override void Info()
      {
         Console.WriteLine("Train");
         Console.WriteLine($"Year: {Year}\n" +
                           $"Weight: {Weight}\n" +
                           $"Color: {Color}");
         Console.WriteLine($"Carriages: {Carriages}\n");
      }
   }
}
using System;
```

```
namespace ConsoleApp1
{
    public class Airplane : Transport
    {
        public double WingLength { get; set; }
        public Airplane(int year, int weight, string color, double wingLength) : base(year, weight,
color)
        {
            WingLength = wingLength;
        }

        public override void Info()
        {
            Console.WriteLine("Airplane");
            Console.WriteLine($"Year: {Year}\n" +
                              $"Weight: {Weight}\n" +
                              $"Color: {Color}");
            Console.WriteLine($"WingLength: {WingLength:0.00}\n");
        }
    }
}
using System;
namespace ConsoleApp1
{
    public class Car : Transport
    {
        public double Speed { get; set; }
        public Car(int year, int weight, string color, double speed) : base(year, weight, color)
        {
            Speed = speed;
        }

        public override void Info()
        {
            Console.WriteLine("Car");
            Console.WriteLine($"Year: {Year}\n" +
                              $"Weight: {Weight}\n" +
                              $"Color: {Color}");
            Console.WriteLine($"Speed: {Speed:0.00}\n");
        }
    }
}
```

Мы успешно создали 3 класса. Добавили поле *Speed* для *Car*, *WingLength* для *Airplane*, *Carriages* для *Train*, реализовали абстрактный метод класса *Transport*.
Так как классы очень походи давайте разберем только один, например Car.

```
public class Car : Transport
```

Этот синтаксис означает что мы публично унаследовали класс родителя *Transport*. Также унаследовали поля родителя:

```
public Car(int year, int weight, string color, double speed)
  : base(year, weight, color)
```

Далее переопределили метод Info() также родителя. Ключевое слово override означает что мы как раз это и сделали.

```
public override void Info()
```

3) Теперь давайте создадим классы и унаследуем их от родителя Auto *“Легковое авто”, “Грузовое авто”:*

```
using System;

namespace ConsoleApp1
{
   public class Truck : Car
   {
      public double BodyLength { get; set; }
      public Truck(int year, int weight, string color, double speed, double bodyLength) :
base(year, weight, color, speed)
      {
         BodyLength = bodyLength;
      }

      public override void Info()
      {
         Console.WriteLine("Truck");
         Console.WriteLine($"Year: {Year}\n" +
                           $"Weight: {Weight}\n" +
                           $"Color: {Color}\n" +
                           $"Speed: {Speed:0.00}");
         Console.WriteLine($"BodyLength: {BodyLength:0.00}\n");
      }
   }
}
using System;

namespace ConsoleApp1
{
   public class Passenger : Car
   {
      public string PassengerType { get; set; }
      public Passenger(int year, int weight, string color, double speed, string passengerType) :
base(year, weight, color, speed)
      {
         PassengerType = passengerType;
      }

      public override void Info()
      {
         Console.WriteLine("Truck");
```

```
            Console.WriteLine($"Year: {Year}\n" +
                              $"Weight: {Weight}\n" +
                              $"Color: {Color}\n" +
                              $"Speed: {Speed:0.00}");
            Console.WriteLine($"PassengerType: {PassengerType}\n");
        }
    }
}
```

Тут ничего сложного, все по аналогии. Теперь нужно создать последние классы: *“Грузовой самолет” и “Пассажирский самолет”:*

```
using System;

namespace ConsoleApp1
{
    public class PassengerPlane : Airplane
    {
        public int Seats { get; set; }
        public PassengerPlane(int year, int weight, string color, double wingLength, int seats) :
base(year, weight, color, wingLength)
        {
            Seats = seats;
        }

        public override void Info()
        {
            Console.WriteLine("Airplane");
            Console.WriteLine($"Year: {Year}\n" +
                              $"Weight: {Weight}\n" +
                              $"Color: {Color}\n" +
                              $"WingLength: {WingLength:0.00}");
            Console.WriteLine($"Seats: {Seats}\n");
        }
    }
}
using System;

namespace ConsoleApp1
{
    public class CargoPlane : Airplane
    {
        public double Capacity { get; set; }
        public CargoPlane(int year, int weight, string color, double wingLength, double capacity)
: base(year, weight, color, wingLength)
        {
            Capacity = capacity;
        }

        public override void Info()
        {
            Console.WriteLine("Airplane");
```

```
            Console.WriteLine($"Year: {Year}\n" +
                              $"Weight: {Weight}\n" +
                              $"Color: {Color}\n" +
                              $"WingLength: {WingLength:0.00}");
            Console.WriteLine($"Capacity: {Capacity:0.00}\n");
        }
    }
}
```

Тут также все по антологии.
Вот и все что нужно было сделать. Теперь давайте проверим все ли работает. Создадим объекты классов:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {

        static void Main(string[] args)
        {
            Random rand = new Random();
            string[] colors = new string[] { "Blue", "Black", "Red", "Green", "Yellow" };
            string[] passengerType = new string[] { "WithBody", "WithoutBody" };

            Transport[] arr = new Transport[10];

            for (int i = 0; i < arr.Length; i++)
            {
                switch (rand.Next(5))
                {
                    case 0:
                        string colorPassanger = colors[rand.Next(0, colors.Length)];
                        string type = passengerType[rand.Next(0, passengerType.Length)];
                        arr[i] = new Passenger(rand.Next(18, 45), rand.Next(500), colorPassanger,
rand.NextDouble() * 30 + 210, type);
                        break;
                    case 1:
                        string colorTruck= colors[rand.Next(0, colors.Length)];
                        arr[i] = new Truck(rand.Next(20, 50), rand.Next(5000), colorTruck,
rand.NextDouble() * 10 + 120, rand.NextDouble() * 2 + 3);
                        break;
                    case 2:
                        string colorCargoPlane = colors[rand.Next(0, colors.Length)];
                        arr[i] = new CargoPlane(rand.Next(60, 150), rand.Next(50000), colorCargoPlane,
rand.NextDouble() * 5 + 10, rand.NextDouble() * 5000 + 7000);
                        break;
```

```
                case 3:
                    string colorPassengerPlane = colors[rand.Next(0, colors.Length)];
                    arr[i] = new PassengerPlane(rand.Next(60, 150), rand.Next(50000), colorPassengerPlane, rand.NextDouble() * 5 + 10, rand.Next(120));
                    break;
                case 4:
                    string colorTrain = colors[rand.Next(0, colors.Length)];
                    arr[i] = new Train(rand.Next(150, 250), rand.Next(100000), colorTrain, rand.Next(10));
                    break;
            }
        }
        foreach (var transport in arr)
        {
            transport.Info();
        }
    }
}
```