ЛЕКЦИЯ 3В. КЛАССЫ И ОБЪЕКТЫ В ЯЗЫКЕ C#. НАСЛЕДОВАНИЕ

В данной теме мы продолжим работу над классами, созданными на прошлом занятии.

ОСНОВНЫЕ ВОПРОСЫ, КОТОРЫЕ РАССМАТРИВАЮТСЯ В ЛЕКЦИИ:

ПОЛНЫЙ ТЕКСТ

1. Описание класса-потомка

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия. Если имя предка не указано, предком считается базовый класс всей иерархии **System.Object**:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
      тело класса
```

Обратите внимание, что слово «предки» присутствует в описании класса во множественном числе, хотя класс может иметь только одного предка. Причина в том, что класс наряду с единственным предком может наследовать от интерфейсов — специального вида классов, не имеющих реализации. Интерфейсы будут рассматриваться на следующей лекции.

Рассмотрим наследование классов на примере. Ранее был описан класс **Monster**, моделирующий персонаж компьютерной игры. Допустим, нам требуется ввести в игру еще один тип персонажей, который должен обладать свойствами объекта **Monster**, а кроме того, уметь думать. Будет логично сделать новый объект потомком объекта **Monster** (листинг 3.1).

Листинг 3.1. Класс Daemon, потомок класса Monster

```
using System;
namespace ConsoleApplication1
```

```
{
   class Monster
   {
      ...
   }

   class Daemon : Monster
   {
      public Daemon()
      {
         brain = 1;
      }

      public Daemon( string name, int brain ) : base( name )     // 1
      {
         this.brain = brain;
      }
      public Daemon( int health, int ammo, string name, int brain )
              : base( health, ammo, name )                        // 2
      {
         this.brain = brain;
      }
      new public void Passport()                                  // 3
      {
         Console.WriteLine(
               "Демон {0} \t здоровье = {1} оружие = {2} ум = {3}",
               Name, Health, Ammo, brain );
      }
      public void Think()                                         // 4
      {
         Console.Write( Name + " сейчас" );
         for ( int i = 0; i < brain; ++i )
            Console.Write( " думает" );
         Console.WriteLine( "..." );
      }
      int brain;          // закрытое поле
   }

   class Class1
   {
      static void Main()
      {
         Daemon Dima = new Daemon("Дима", 3 );                    // 5
         Dima.Passport();                                         // 6
         Dima.Think();                                            // 7
         Dima.Health -= 10;                                       // 8
         Dima.Passport();
      }
   }
}
```

В классе **Daemon** введены закрытое поле **brain** и метод **Think**, определены собственные конструкторы, а также переопределен метод **Passport**. Все поля и свойства класса **Monstr** наследуются в классе **Daemon**.

Результат работы программы:

```
Демон Дима      здоровье = 100 оружие = 100 ум = 3
Дима сейчас думает думает думает...
Демон Дима      здоровье = 90 оружие = 100 ум = 3
```

Как видите, экземпляр класса **Daemon** с одинаковой легкостью использует как собственные (операторы 5–7), так и унаследованные (оператор 8) элементы класса. Рассмотрим общие правила наследования.

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

1. Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.
2. Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.
3. Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова **base**. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова **base**.

Поля, ***методы*** и ***свойства*** класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать компилятору свое намерение с помощью ключевого слова **new**. В листинге 3.1 таким образом переопределен метод вывода информации об объекте **Passport**. Метод **Passport** класса **Daemon** замещает соответствующий метод базового класса, однако возможность доступа к методу базового класса из метода производного класса сохраняется. Для этого перед вызовом метода указывается все то же волшебное слово **base**, например:

```
base.Passport();
```

Элементы базового класса, определенные как **private**, в производном классе недоступны. Поэтому в методе **Passport** для доступа к полям **name**, **health** и **ammo** пришлось использовать соответствующие свойства базового класса. Другое решение заключается в том, чтобы определить эти поля со спецификатором **protected**, в этом случае они будут доступны методам всех классов, производных от **Monster**. Оба решения имеют свои достоинства и недостатки.

Во время выполнения программы объекты хранятся в отдельных переменных, массивах или других коллекциях. Во многих случаях удобно ***оперировать объектами одной иерархии единообразно***, то есть использовать один и тот же программный код для работы с экземплярами разных

классов. Это возможно благодаря тому, что ***объекту базового класса можно присвоить объект производного класса***.

Попробуем описать массив объектов базового класса и занести туда объекты производного класса. В листинге 3.2 в массиве типа **Monster** хранятся два объекта типа **Monster** и один — типа **Daemon**.

Листинг 3.2. Массив объектов разных типов

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      ...
   }

   class Daemon : Monster
   {
      ... //
   }

   class Class1
   {
      static void Main()
      {
         const int n = 3;
         Monster[] stado = new Monster[n];

         stado[0] = new Monster( "Маня" );
         stado[1] = new Monster( "Monk" );
         stado[2] = new Daemon ( "Димон", 3 );

         foreach ( Monster elem in stado ) elem.Passport();      // 1
         for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;        // 2
            Console.WriteLine();

         foreach ( Monster elem in stado ) elem.Passport();      // 3
      }
   }
}
```

Результат работы программы:

```
Монстер Маня     здоровье = 100 оружие = 100
Монстер Monk     здоровье = 100 оружие = 100
Монстер Димон    здоровье = 100 оружие = 100
Монстер Маня     здоровье = 100 оружие = 0
Монстер Monk     здоровье = 100 оружие = 0
Монстер Димон    здоровье = 100 оружие = 0
```

Результат радует нас только частично: объект типа **Daemon** действительно можно поместить в массив, состоящий из элементов типа **Monster**, но для него вызываются только методы и свойства, унаследованные от предка. Это устраивает нас в операторе 2, а в операторах 1 и 3 хотелось бы, чтобы вызывался

метод **Passport**, переопределенный в потомке.

Итак, присваивать объекту базового класса объект производного класса можно, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает.

Это и понятно: ведь компилятор должен еще до выполнения программы решить, какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод (этот процесс называется ***ранним связыванием***). При этом компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство (например, **stado[i].Ammo**). То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее — до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка. Такой механизм в C# есть — он называется ***поздним связыванием*** и реализуется с помощью так называемых виртуальных методов, которые мы незамедлительно и рассмотрим.

2. Виртуальные методы

При раннем связывании программа, готовая для выполнения, представляет собой структуру, логика выполнения которой жестко определена. Если же требуется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного объекта, для которого выполняется вызов, то заранее жестко связывать эти методы с остальной частью кода нельзя.

Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться по-другому. Для этого в C# существует ключевое слово **virtual**. Оно записывается в заголовке метода базового класса, например:

```
virtual public void Passport() ...
```

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться в момент его вызова во время выполнения программы. Этот механизм называется ***поздним связыванием***.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов ***таблицу виртуальных методов*** (Virtual Method Table, VMT). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.

Каждый объект во время выполнения должен иметь доступ к VMT. Связь экземпляра объекта с VMT устанавливается с помощью специального кода, автоматически помещаемого компилятором в конструктор объекта.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово **override**, например:

```
override public void Passport() ...
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.

Добавим в листинг класса **Monster** два волшебных слова — **virtual** и **override** — в описания методов **Passport** соответственно базового и производного классов (листинг 3.3).

Листинг 3.3. Виртуальные методы

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      ...
      virtual public void Passport()
      {
         Console.WriteLine("Монстер {0}\t здоровье={1} оружие={2}",
                           name, health, ammo );
       }
       ...
   }

   class Daemon : Monster
   {
      ...
      override public void Passport()
      {
         Console.WriteLine(
             "Демон {0} \t здоровье = {1} оружие = {2} ум = {3}",
              Name, Health, Ammo, brain );
      }
      ...
   }

   class Class1
   {
      static void Main()
      {
         const int n = 3;
         Monster[] stado = new Monster[n];

         stado[0] = new Monster( "Маня" );
         stado[1] = new Monster( "Monk" );
         stado[2] = new Daemon ( "Димон", 3 );

         foreach ( Monster elem in stado ) elem.Passport();

         for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
         Console.WriteLine();

         foreach ( Monster elem in stado ) elem.Passport();
      }
   }
```

```
}
```

Результат работы программы:

```
Монстер Маня      здоровье = 100 оружие = 100
Монстер Monk      здоровье = 100 оружие = 100
Демон Димон       здоровье = 100 оружие = 100 ум = 3
Монстер Маня      здоровье = 100 оружие = 0
Монстер Monk      здоровье = 100 оружие = 0
Демон Димон       здоровье = 100 оружие = 0 ум = 3
```

Теперь в циклах 1 и 3 вызывается метод **Passport**, соответствующий типу объекта, помещенного в массив.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — ***полиморфизм***. Это слово в переводе с греческого означает «много форм», что в данном случае означает «один вызов — много методов».

Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызове в нее передается объект производного класса. Виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

3. Абстрактные классы

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.

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором **abstract**.

Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например:

```
abstract class Spirit
{
   public abstract void Passport();
```

```
}

class Monster : Spirit
{
   ...
   override public void Passport()
   {
      Console.WriteLine("Монстер {0}\t здоровье = {1} оружие = {2}",
                        name, health, ammo );
   }
   ...
}

class Daemon : Monster
{
   ...
   override public void Passport()
   {
      Console.WriteLine(
             "Демон {0} \t здоровье = {1} оружие = {2} ум = {3}",
              Name, Health, Ammo, brain );
   }
   ... // полный текст этих классов
}
```

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов.

4. Бесплодные классы

В C# есть ключевое слово **sealed**, позволяющее описать класс, от которого, в противоположность абстрактному, наследовать запрещается:

```
sealed class Spirit
{
   ...
}
//  class Monster : Spirit { ... }          ошибка!
```

Большинство встроенных типов данных описано как sealed. Если необходимо использовать функциональность бесплодного класса, применяется не наследование, а ***вложение***, или ***включение***: в классе описывается поле соответствующего типа.

Вложение классов, когда один класс включает в себя поля, являющиеся классами, является ***альтернативой наследованию*** при проектировании. Например, если есть объект «двигатель», а требуется описать объект «самолет», логично сделать двигатель полем этого объекта, а не его предком.

5. Класс object

Корневой класс **System.Object** всей иерархии объектов .NET, называемый в C# **object**, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Класс **object** часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется полиморфизм.

Открытые методы класса **System.Object** перечислены ниже.

1. Метод **Equals** с одним параметром возвращает значение **true**, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

```
public virtual bool Equals(object obj);
```

2. Метод **Equals** с двумя параметрами возвращает значение **true**, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool Equals(object ob1, object ob2);
```

3. Метод **GetHashCode** формирует хэш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод **Equals**, необходимо перегрузить и метод **GetHashCode**. Синтаксис:

```
public virtual int GetHashCode();
```

4. Метод **GetType** возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип Type. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения. Синтаксис:

```
public Type GetType();
```

5. Метод **ReferenceEquals** возвращает значение **true**, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool ReferenceEquals(object ob1, object ob2);
```

6. Метод **ToString** по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис:

```
public virtual string ToString()
```

В производных объектах эти методы часто переопределяют. Например, можно переопределить метод **Equals** для того, чтобы задать собственные

критерии сравнения объектов.

Пример применения и переопределения методов класса **object** для класса **Monster** приведен в листинге 3.4.

Листинг 3.4. Перегрузка методов класса object

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }

      public override bool Equals( object obj )
      {
         if (obj==null || GetType()!=obj.GetType()) return false;
         Monster temp = (Monster) obj;
         return  health == temp.health &&
                 ammo   == temp.ammo   &&
                 name   == temp.name;
      }

      public override int GetHashCode()
      {
         return name.GetHashCode();
      }

      public override string ToString()
      {
       return string.Format("Монстер {0}\t здоровье={1} оружие={2}",
                 name, health, ammo );
      }

      string name;
      int health, ammo;
   }

   class Class1
   {  static void Main()
      {
         Monster X = new Monster( 80, 80, "Вася" );
         Monster Y = new Monster( 80, 80, "Вася" );
         Monster Z = X;

         if ( X == Y ) Console.WriteLine(" X == Y ");
         else          Console.WriteLine(" X != Y ");

         if ( X == Z ) Console.WriteLine(" X == Z ");
         else          Console.WriteLine(" X != Z ");
```

```
            if ( X.Equals(Y) ) Console.WriteLine( " X Equals Y " );
            else               Console.WriteLine( " X not Equals Y " );

            Console.WriteLine(X.GetType());
        }
    }
}
```

Результат работы программы:

```
X != Y
X == Z
X Equals Y
ConsoleApplication1.Monster
```

Анализируя результат работы программы, можно увидеть, что в операции сравнения на равенство сравниваются ссылки, а в перегруженном методе **Equals** — значения. Для концептуального единства можно переопределить и операции отношения.

Листинги для работы на занятии

1. **Класс Daemon, потомок класса Monster**:

```
class Daemon : Monster
{
   int brain;         // закрытое поле - ум

   public void Think() // Новый метод - думать
   {
      Console.Write(Name + " сейчас");
      for (int i = 0; i < brain; ++i)
         Console.Write(" думает");
      Console.WriteLine("...");
   }

   new public void Passport()  // Переопределенный метод
   {
      Console.WriteLine(
         "Демон {0} \t здоровье = {1} оружие = {2} ум = {3}",
          Name, Health, Ammo, brain);
   }

   // конструкторы
   public Daemon()
   {
      brain = 1;
   }

   public Daemon(string name, int brain) : base(name)
   {
      this.brain = brain;
   }
```

```
    public Daemon(int health, int ammo, string name, int brain)
         : base(health, ammo, name)
    {
        this.brain = brain;
    }
}
```

2. ПРОВЕРКА РАБОТЫ: Класс Daemon

```
class Program
{
    static void Main(string[] args)
    {
        Monster Vasia = new Monster("Вася");
        Vasia.Passport();
        Daemon Dima = new Daemon("Дима", 3);
        Dima.Passport();
        Dima.Think();
        Dima.Health -= 10;
        Dima.Passport();
        Console.ReadKey();
    }
}
```

3. ПРОВЕРКА РАБОТЫ: МАССИВ из МОНСТРОВ и ДЕМОНОВ

```
class Program
{
    static void Main(string[] args)
    {
        const int n = 3;
        Monster[] stado = new Monster[n];

        stado[0] = new Monster( "Маня" );
        stado[1] = new Monster( "Monk" );
        stado[2] = new Daemon ( "Димон", 3 );

        foreach ( Monster elem in stado ) elem.Passport();     // 1

        for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;       // 2

        Console.WriteLine();

        foreach ( Monster elem in stado ) elem.Passport();      // 3
    }
}
```

4. Виртуальные методы

```
class Monster
{
    // ЗДЕСЬ НАХОДИТСЯ ВСЯ РЕАЛИЗАЦИЯ КЛАССА,
    // КОТОРАЯ ЕСТЬ НА ДАННЫЙ МОМЕНТ!!!
```

```
    // Виртуальный метод
    virtual public void Passport()
    {
        // Паспортные данные
        Console.WriteLine("У монстра {0} \t Здоровье={1} Оружие={2}
              Состояние:{3}", name, health, ammo, GetState());
    }
}
class Daemon : Monster
{
    // ЗДЕСЬ НАХОДИТСЯ ВСЯ РЕАЛИЗАЦИЯ КЛАССА,
    // КОТОРАЯ ЕСТЬ НА ДАННЫЙ МОМЕНТ!!!

    // Переопределенный виртуальный метод
    override public void Passport()
    {
        Console.WriteLine(
             "Демон {0} \t здоровье = {1} оружие = {2} ум = {3}",
              Name, Health, Ammo, brain);
    }
}
```

5. ВКЛЮЧЕНИЕ классов

```
class Arsenal // оружие
{
    int bullet; // количество пуль
    public Arsenal()
    {
        bullet = 7;
    }
    public void Fire()
    {
        if (bullet>0)
        {
            Console.WriteLine("БАХ-БАХ!!!");
            bullet--;
         }
          else
             Console.WriteLine("Кончились заряды! Перезарядите
меня!!!");
    }
    public void Reload(int b)
    {
        if (b >= 0 && b <= 7) bullet = b;
        if (b > 7) bullet = 7;
        if (b < 0) bullet = 0;
    }
}

class Monster
{   // ПОЛЯ
    private string name; // имя монстра
    private int health; // здоровье
```

```
    private int ammo;     // оружие
    public Arsenal arsenal;  // пистолет

    public Monster()      // конструктор
    {
        this.name = "Noname";
        this.health = 100;
        this.ammo = 100;
        arsenal = new Arsenal();
    }

    // МЕТОДЫ
    public void Bax()
    {
        arsenal.Fire();
    }

    public void Reload(int k)
    {
        arsenal.Reload(k);
    }
}
```

6. ПРОВЕРКА РАБОТЫ: ВКЛЮЧЕНИЕ классов

```
class Program
{
    static void Main(string[] args)
    {
        Monster Vasya = new Monster();
        for(int i=0; i<10; i++)
            Vasya.Bax();
        Vasya.Reload(10);
        for (int i = 0; i < 10; i++)
            Vasya.Bax();
        Console.ReadKey();
    }
}
```

7. ПРОВЕРКА МЕТОДОВ КЛАССА object

```
class Program
{
    static void Main(string[] args)
    {
        Monster X = new Monster(80, 80, "Вася");
        Monster Y = new Monster(80, 80, "Вася");
        Monster Z = X;

        if (X == Y) Console.WriteLine(" X == Y ");
        else Console.WriteLine(" X != Y ");

        if (X == Z) Console.WriteLine(" X == Z ");
        else Console.WriteLine(" X != Z ");
```

```
        if (X.Equals(Y)) Console.WriteLine(" X Equals Y ");
        else Console.WriteLine(" X not Equals Y ");

        if (X.Equals(Z)) Console.WriteLine(" X Equals Z ");
        else Console.WriteLine(" X not Equals Z ");

        Console.WriteLine(X.GetType());
        Console.WriteLine(X.ToString());

        Console.ReadKey();
    }
}
```

Перегрузка методов класса Object

```
class Monster
{
    // Перегрузка методов класса object
    public override bool Equals(object obj)
    {
        if (obj == null || GetType() != obj.GetType()) return false;
        Monster temp = (Monster)obj;
        return health == temp.health &&
               ammo == temp.ammo &&
               name == temp.name;
    }

    public override int GetHashCode()
    {
        return name.GetHashCode();
    }

    public override string ToString()
    {
        return string.Format("Монстер {0}\t Здоровье={1} Оружие={2}",
               name, health, ammo);
    }
}
```

ПРОВЕРКА РАБОТЫ: ПЕРЕГРУЗКА МЕТОДОВ КЛАССА object

```
class Program
{
    static void Main(string[] args)
    {
        Monster X = new Monster(80, 80, "Вася");
        Monster Y = new Monster(80, 80, "Вася");
        Monster Z = X;

        if (X == Y) Console.WriteLine(" X == Y ");
        else Console.WriteLine(" X != Y ");

        if (X == Z) Console.WriteLine(" X == Z ");
```

```
        else Console.WriteLine(" X != Z ");

        if (X.Equals(Y)) Console.WriteLine(" X Equals Y ");
        else Console.WriteLine(" X not Equals Y ");

        if (X.Equals(Z)) Console.WriteLine(" X Equals Z ");
        else Console.WriteLine(" X not Equals Z ");

        Console.WriteLine(X.GetType());
        Console.WriteLine(X.ToString());

        Console.ReadKey();
    }
}
```