ЛЕКЦИЯ 2. КЛАССЫ И ОБЪЕКТЫ В ЯЗЫКЕ C#. СОЗДАНИЕ КЛАССОВ И ОБЪЕКТОВ. ОСНОВНЫЕ ЭЛЕМЕНТЫ КЛАССОВ

Поскольку язык C# по своей природе является полностью объектно-ориентированным языком программирования, то и его изучение мы начнем с изучения основ создания классов.

ОСНОВНЫЕ ВОПРОСЫ, КОТОРЫЕ РАССМАТРИВАЮТСЯ В ЛЕКЦИИ:

ПОЛНЫЙ ТЕКСТ

1. Описание класса

Класс является типом данных, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. ***Элементами*** класса являются ***данные*** и ***функции***, предназначенные для их обработки.

Описание класса содержит ключевое слово **class**, за которым следует его ***имя***, а далее в фигурных скобках — ***тело*** класса, то есть список его элементов. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
тело_класса
```

Как видите, обязательными являются только ключевое слово **class**, а также имя и тело класса. ***Тело класса*** — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может

выглядеть так:

```
class Demo {}
```

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в таблице 2.1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется ***вложенным***.

Таблица 2.1. Спецификаторы класса

№	Спецификатор	Описание
1	new	Используется для вложенных классов. Задает новое описание класса взамен унаследованного от предка. Применяется в иерархиях объектов
2	public	Доступ не ограничен
3	protected	Используется для вложенных классов. Доступ только из элементов данного и производных классов
4	internal	Доступ только из данной программы (сборки)
5	protected internal	Доступ только из данного и производных классов или из данной программы (сборки)
6	private	Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс
7	abstract	Абстрактный класс. Применяется в иерархиях объектов
8	sealed	Бесплодный класс. Применяется в иерархиях объектов
9	static	Статический класс. Введен в версию языка 2.0.

Спецификаторы 2–6 называются ***спецификаторами доступа***. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут присутствовать в описании только в вариантах, приведенных в таблице, а также могут комбинироваться с остальными спецификаторами.

Сейчас мы будем изучать только классы, которые описываются в пространстве имен непосредственно (то есть не вложенные классы). Для таких классов допускаются два спецификатора: **public** и **internal**. По умолчанию подразумевается **internal**.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых ***экземплярами***, или ***объектами***, ***класса***. Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции **new**, например:

```
Demo a = new Demo();     // создание экземпляра класса Demo
Demo b = new Demo();     // создание другого экземпляра класса Demo
```

Для каждого объекта при его создании в памяти выделяется отдельная

область, в которой хранятся его данные. Кроме того, в классе могут присутствовать ***статические элементы***, которые существуют в единственном экземпляре для всех объектов класса. Часто статические данные называют ***данными класса***, а остальные — ***данными экземпляра***.

Функциональные элементы класса не тиражируются, то есть всегда хранятся в единственном экземпляре. Для работы с данными класса используются ***методы класса*** (***статические методы***), для работы с данными экземпляра — ***методы экземпляра***, или просто ***методы***.

Поля и методы являются основными элементами класса. Кроме того, в классе можно задавать целую гамму других элементов: свойства, события, индексаторы, операции, конструкторы, деструкторы, а также типы (рис. 2.1).

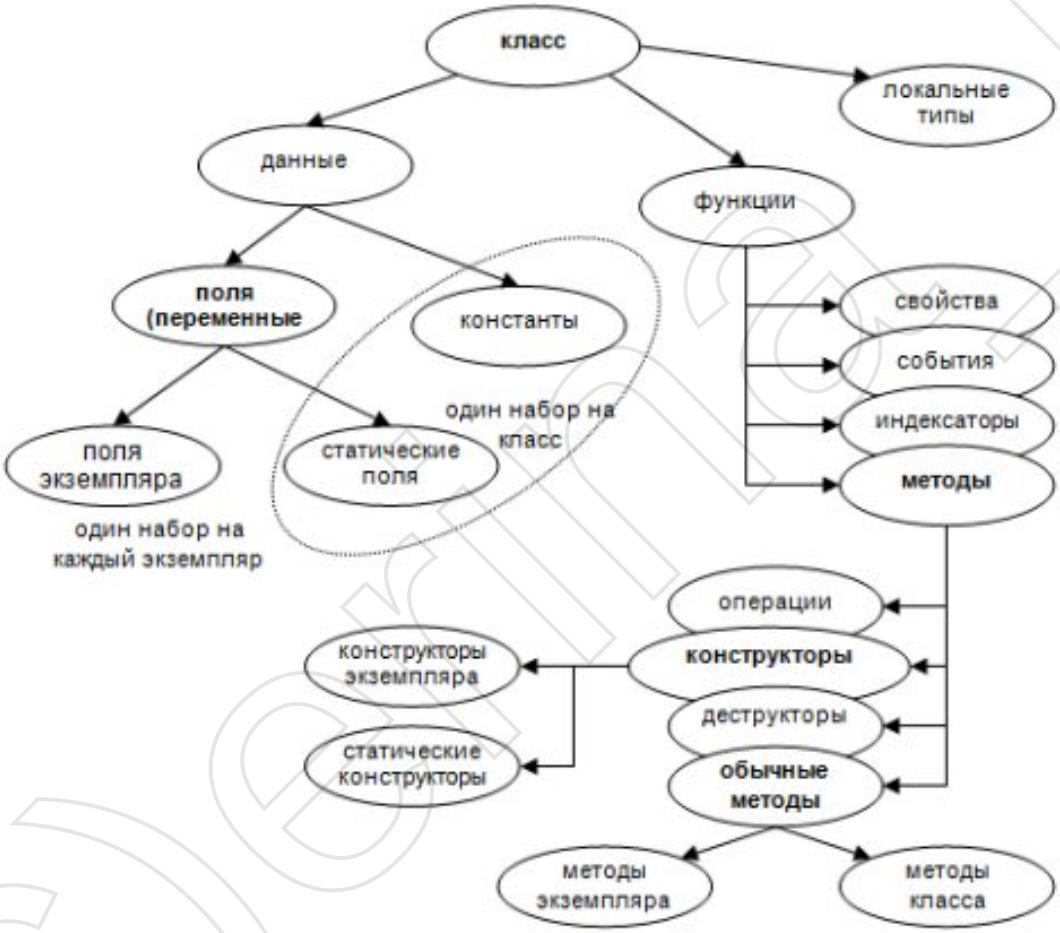

Рис. 2.1. Состав класса

Ниже приведено краткое описание всех элементов класса, изображенных на рисунке:

- ***Константы*** класса хранят неизменяемые значения, связанные с классом.
- ***Поля*** содержат данные класса.
- ***Методы*** реализуют вычисления или другие действия, выполняемые классом или экземпляром.
- ***Свойства*** определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения.
- ***Конструкторы*** реализуют действия по инициализации экземпляров или класса в целом.

– ***Деструкторы*** определяют действия, которые необходимо выполнить до того, как объект будет уничтожен.
– ***Индексаторы*** обеспечивают возможность доступа к элементам класса по их порядковому номеру.
– ***Операции*** задают действия с объектами с помощью знаков операций.
– ***События*** определяют уведомления, которые может генерировать класс.
– ***Типы*** — это типы данных, внутренние по отношению к классу.

Прежде чем начать изучение элементов класса, необходимо поговорить о присваивании и сравнении объектов.

Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и значимого, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки — ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти (рис. 2.2).

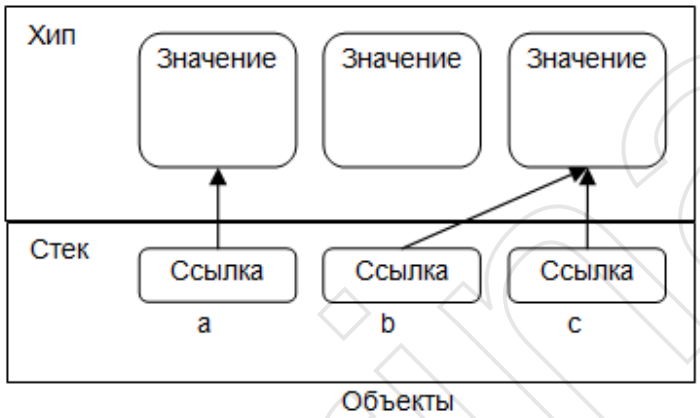

Рис. 2.2. Присваивание объектов

Аналогичная ситуация с операцией проверки на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке объекты **b** и **c** равны, но **a** не равно **b** даже при равенстве их значений или если они обе равны **null**).

2. Данные: поля и константы

Данные, содержащиеся в классе, могут быть ***переменными*** или ***константами*** и задаются в соответствии с правилами для обычных переменных языка и именованных констант.

Переменные, описанные в классе, называются ***полями*** класса. При описании элементов класса можно также указывать атрибуты и спецификаторы, задающие различные характеристики элементов. Синтаксис описания элемента данных:

```
[атрибуты] [спецификаторы] [const] тип имя [= начальное_значение]
```

Возможные ***спецификаторы*** полей и констант перечислены в таблице 2.2. Для констант можно использовать только спецификаторы 1–6.

Таблица 2.2. Спецификаторы полей и констант класса

№	Спецификатор	Описание
1	new	Новое описание поля, скрывающее унаследованный элемент класса
2	public	Доступ к элементу не ограничен
3	protected	Доступ только из данного и производных классов
4	internal	Доступ только из данной сборки
5	protectedinternal	Доступ только из данного и производных классов и из данной сборки
6	private	Доступ только из данного класса
7	static	Одно поле для всех экземпляров класса
8	readonly	Поле доступно только для чтения
9	volatile	Поле может изменяться другим процессом или системой

По умолчанию элементы класса считаются **закрытыми** (**private**). Для полей класса этот вид доступа является предпочтительным. Все методы класса имеют непосредственный доступ к его закрытым полям.

Поля, описанные со спецификатором **static**, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Если класс содержит только статические элементы, экземпляр класса создавать не требуется.

Обращение к полю класса выполняется с помощью ***операции доступа*** (точка). Справа от точки задается имя поля, слева — имя экземпляра для обычных полей или имя класса для статических. В листинге 2.1 приведен пример простого класса **Demo** и два способа обращения к его полям.

Листинг 2.1. Класс Demo, содержащий поля и константу

```
using System;
namespace ConsoleApplication1
{
   class Demo
   {
      public int a = 1;                  // поле данных
      public const double  c = 1.66;     // константа
      public static string s = "Demo";   // статическое поле класса
      double y;                          // закрытое поле данных
   }

   class Class1
   {
      static void Main()
      {
         Demo x = new Demo();   // создание экземпляра класса Demo
         Console.WriteLine(x.a); // x.a - обращение к полю класса
         Console.WriteLine(Demo.c); //Demo.c-обращение к константе
         Console.WriteLine(Demo.s);//обращение к статическому полю
      }
   }
```

```
}
```

Поля со спецификатором **readonly** предназначены только для чтения. Установить значение такого поля можно либо при его описании, либо в конструкторе (конструкторы рассматриваются далее).

3. Методы

3.1 Понятие метода

Метод — это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или экземпляром. Методы определяют поведение класса.

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо. Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

Синтаксис метода:

```
[ атрибуты ] [ спецификаторы ] тип имя_метода ( [ параметры ] )
тело_метода
```

Первая строка представляет собой ***заголовок*** метода. ***Тело метода***, задающее действия, выполняемые методом, чаще всего представляет собой блок.

При описании методов можно использовать спецификаторы 1–7 из таблицы 2.2, имеющие тот же смысл, что и для полей, а также спецификаторы **virtual**, **sealed**, **override**, **abstract** и **extern**, которые будут рассмотрены по мере необходимости. Чаще всего для методов задается спецификатор доступа **public**, ведь методы составляют интерфейс класса — то, с чем работает пользователь.

Пример простейшего метода:

```
public double Gety()     // метод для получения поля y
{
    return y;
}
```

Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин «метод возвращает значение». Если метод не возвращает никакого значения, в его заголовке задается тип **void**, а оператор **return** отсутствует.

Параметры используются для обмена информацией с методом. Параметр представляет собой локальную переменную, которая при вызове метода принимает значение соответствующего аргумента. Область действия параметра — весь метод.

Например, чтобы вычислить значение синуса для вещественной величины **x**, мы передаем ее в качестве аргумента в метод **Sin** класса **Math**, а чтобы вывести значение этой переменной на экран, мы передаем ее в метод **WriteLine** класса **Console**:

```
double x = 0.1;
double y = Math.Sin(x);
Console.WriteLine(x);
```

При этом метод **Sin** возвращает в точку своего вызова вещественное значение синуса, которое присваивается переменной **y**, а метод **WriteLine** ничего не возвращает.

Метод, не возвращающий значение, вызывается отдельным оператором, а метод, возвращающий значение, — в составе выражения в правой части оператора присваивания.

Параметры, описываемые в заголовке метода, определяют множество значений ***аргументов***, которые можно передавать в метод. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу. Для каждого параметра должны задаваться его ***тип*** и ***имя***. Например, заголовок метода **Sin** выглядит следующим образом:

```
public static double Sin( double a );
```

Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой ***сигнатуру метода***. В классе не должно быть методов с одинаковыми сигнатурами.

В листинге 2.2 в класс **Demo** добавлены методы установки и получения значения поля **y**.

Листинг 2.2. Простейшие методы

```
using System;
namespace ConsoleApplication1
{
   class Demo
   {
      public int a = 1;
      public const double c = 1.66;
      static string s = "Demo";
      double y;

      public double Gety()             // метод получения поля y
      {
         return y;
      }

      public void Sety( double y_ )    // метод установки поля y
      {
         y = y_;
      }

      public static string Gets()      // метод получения поля s
      {
         return s;
      }
   }
```

```
   class Class1
   {
      static void Main()
      {
         Demo x = new Demo();
         x.Sety(0.12);            // вызов метода установки поля y
         Console.WriteLine(x.Gety());//вызов метода получения поля y
         Console.WriteLine(Demo.Gets()); // вызов метода получения
поля s
  //     Console.WriteLine(Gets());  // вариант вызова
      }
   }
}
```

3.2 Параметры методов

При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их).
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип void, управление передается на оператор, следующий после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение. Листинг 2.3 иллюстрирует этот процесс.

Листинг 2.3. Передача параметров методу

```
using System;
namespace ConsoleApplication1
{
   class Class1
   {
    static int Max(int a,int b)//метод выбора максимального значения
      {
        if ( a > b ) return a;
        else         return b;
      }

      static void Main()
      {
         int a = 2, b = 4;
         int x = Max( a, b );                 // вызов метода Max
         Console.WriteLine( x );              // результат: 4
         short t1 = 3, t2 = 4;
         int y = Max( t1, t2 );               // вызов метода Max
         Console.WriteLine( y );              // результат: 4
         int z = Max( a + t1, t1 / 2 * b );   // вызов метода Max
         Console.WriteLine( z );              // результат: 5
```

```
        }
    }
}
```

Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и в заголовке метода, и должно существовать неявное преобразование типа каждого аргумента к типу соответствующего параметра. Количество аргументов должно соответствовать количеству параметров.

Существуют два способа передачи параметров: по значению и по ссылке.

При передаче по значению метод получает копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а, следовательно, нет и возможности их изменить.

При передаче по ссылке (по адресу) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов, модифицируя параметры.

В C# для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

- параметры-значения;
- параметры-ссылки — описываются с помощью ключевого слова ref;
- выходные параметры — описываются с помощью ключевого слова out;
- параметры-массивы — описываются с помощью ключевого слова params.

Ключевое слово предшествует описанию типа параметра. Если оно опущено, параметр считается параметром-значением. Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Calculate( int a, ref int b, out int c, params int[] d )
```

3.3 Параметры-значения

Параметр-значение описывается в заголовке метода следующим образом:

```
тип имя
```

Пример заголовка метода, имеющего один параметр-значение целого типа:

```
void P( int x )
```

Имя параметра может быть произвольным. Параметр x представляет собой локальную переменную, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента.

Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется ее значение и копируется в специальную область памяти — область параметров. Метод работает с этой копией. По завершении работы метода область параметров освобождается. Этот способ годится только для передачи в метод исходных данных.

При вызове метода на месте параметра, передаваемого по значению, может находиться выражение, для типа которого существует неявное преобразование

типа выражения к типу параметра.

Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int    x = 1;
sbyte  c = 1;
ushort y = 1;
```

Тогда следующие вызовы метода P, заголовок которого был описан ранее, будут синтаксически правильными:

```
P( x );    P( c );    P( y );    P( 200 );    P( x / 4 + 1 );
```

3.4 Параметры-ссылки

Признаком параметра-ссылки является ключевое слово ref перед описанием параметра:

```
ref тип имя
```

Пример заголовка метода, имеющего один параметр-ссылку целого типа:

```
void P( ref int x )
```

При вызове метода в область параметров копируется адрес аргумента, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке.

При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово ref.

Проиллюстрируем передачу параметров-значений и параметров-ссылок на примере (листинг 2.4).

Листинг 2.4. Параметры-значения и параметры-ссылки

```
using System;
namespace ConsoleApplication1
{
   class Class1
   {
      static void P( int a, ref int b )
      {
         a = 44; b = 33;
         Console.WriteLine( "внутри метода {0} {1}", a, b );
      }

      static void Main()
      {
         int a = 2, b = 4;
         Console.WriteLine( "до вызова     {0} {1}", a, b );
```

```
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

Результаты работы этой программы:

```
до вызова     2  4
внутри метода 44 33
после вызова  2  33
```

Несколько иная картина получится, если передавать в метод не величины значимых типов, а ***экземпляры классов***, то есть величины ссылочных типов. Для простоты можно считать, что объекты всегда передаются по ссылке.

3.5 Выходные параметры

Довольно часто возникает необходимость в методах, которые формируют несколько величин. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор **out**. Параметру, имеющему этот спецификатор, должно быть обязательно присвоено значение внутри метода.

Изменим описание второго параметра в листинге 2.4 так, чтобы он стал ***выходным*** (листинг 2.5).

Листинг 2.5. Выходные параметры

```
using System;
namespace ConsoleApplication1
{
   class Class1
   {
      static void P( int a, out int b )
      {
         a = 44; b = 33;
         Console.WriteLine( "внутри метода {0} {1}", a, b );
      }

      static void Main()
      {
         int a = 2, b;
         P( a, out b );
         Console.WriteLine( "после вызова  {0} {1}", a, b );
      }
   }
}
```

При вызове метода перед соответствующим параметром тоже указывается ключевое слово **out**.

3.6 Ключевое слово this

Каждый объект содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию экземпляр.

В явном виде параметр **this** применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
class Demo
{
   double y;

   public Demo T()          // метод возвращает ссылку на экземпляр
   {
      return this;
   }

   public void Sety( double y )
   {
      this.y = y;        // полю y присваивается значение параметра y
   }
}
```

4. Конструкторы

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции **new**. Имя конструктора совпадает с именем класса. Ниже перечислены свойства конструкторов.

- Конструктор ***не возвращает значение***, даже типа **void**.
- Класс может иметь ***несколько конструкторов*** с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение null.
- Конструктор, вызываемый без параметров, называется ***конструктором по умолчанию***.

До сих пор мы задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать в конструкторе. В листинге 2.6 в класс **Demo** добавлен конструктор, а поля сделаны закрытыми.

Листинг 2.6. Класс с конструктором

```
using System;
namespace ConsoleApplication1
{
   class Demo
   {
      public Demo( int a, double y )     // конструктор с параметрами
      {
         this.a = a;
         this.y = y;
      }

      public double Gety()               // метод получения поля y
      {
         return y;
      }

      int a;
      double y;
   }
   class Class1
   {
      static void Main()
      {
         Demo a = new Demo( 300, 0.002 );      // вызов конструктора
         Console.WriteLine( a.Gety() );        // результат: 0,002
         Demo b = new Demo( 1, 5.71 );         // вызов конструктора
         Console.WriteLine( b.Gety() );        // результат: 5,71
      }
   }
}
```

Часто бывает удобно задать в классе *несколько конструкторов*, чтобы обеспечить возможность инициализации объектов разными способами. Все конструкторы должны иметь разные сигнатуры.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, удобно *вызвать первый конструктор из второго*. Для этого используется уже известное вам ключевое слово **this** в другом контексте, например:

```
   class Demo
   {
      public Demo( int a )                          // конструктор 1
      {
         this.a = a;
      }

      public Demo(int a, double y : this(a)  // вызов конструктора 1
      {
         this.y = y;
      }
      ...
   }
```

Конструкция, находящаяся после двоеточия, называется *инициализатором*.

Как вы помните, все классы в C# имеют общего предка — класс **object**. Конструктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка.

До сих пор речь шла об «обычных» конструкторах, или *конструкторах экземпляра*. Существует второй тип конструкторов — *статические конструкторы*, или *конструкторы класса*. Конструктор экземпляра инициализирует данные экземпляра, конструктор класса — данные класса.

Статический конструктор не имеет параметров, его нельзя вызвать явным образом. Система сама определяет момент, в который требуется его выполнить.

Некоторые классы содержат только статические данные и, следовательно, создавать экземпляры таких объектов не имеет смысла. В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором **static**. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором **static** (константы и вложенные типы классифицируются как статические элементы автоматически). В листинге 2.7 приведен пример статического класса.

Листинг 2.7. Статический класс (начиная с версии 2.0)

```
using System;
namespace ConsoleApplication1
{
   static class D
   {
      static int a = 200;
      static double b = 0.002;
      public static void Print ()
      {
         Console.WriteLine( "a = " + a );
         Console.WriteLine( "b = " + b );
      }
   }
   class Class1
   {  static void Main()
      {
         D.Print();
      }
   }
}
```

В качестве «сквозного» примера, на котором будет демонстрироваться работа с различными элементами класса, создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупальцев или наличие гранатомета) и поведение.

Листинг 2.8. Класс Monster

```
using System;
namespace ConsoleApplication1
```

```
{
   class Monster
   {
      public Monster()
      {
         this.name   = "Noname";
         this.health = 100;
         this.ammo   = 100;
      }
      public Monster( string name ) : this()
      {
         this.name = name;
      }
      public Monster( int health, int ammo, string name )
      {
         this.name   = name;
         this.health = health;
         this.ammo   = ammo;
      }

      public int GetName()
      {
         return name;
      }
      public int GetHealth()
      {
         return health;
      }
      public int GetAmmo()
      {
         return ammo;
      }
      public void Passport()
      {
         Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
                           name, health, ammo );
      }
      string name;                // закрытые поля
      int health, ammo;
   }

   class Class1
   {
      static void Main()
      {
         Monster X = new Monster();
         X.Passport();
         Monster Vasia = new Monster( "Vasia" );
         Vasia.Passport();
         Monster Masha = new Monster( 200, 200, "Masha" );
         Masha.Passport();
      }
   }
}
```

Результат работы программы:

```
Monster Noname   health = 100 ammo = 100
Monster Vasia    health = 100 ammo = 100
Monster Masha    health = 200 ammo = 200
```

5. Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. Синтаксис свойства:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
      [ get код_доступа ]
      [ set код_доступа ]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются со спецификатором **public**. *Код доступа* представляет собой блоки операторов, которые выполняются при получении (**get**) или установке (**set**) свойства. Может отсутствовать либо часть **get**, либо **set**, но не обе одновременно.

Если отсутствует часть **set**, свойство доступно только для чтения (***read-only***), если отсутствует часть **get**, свойство доступно только для записи (***write-only***). В версии C# 2.0 введена возможность задавать разный уровень доступа для частей **get** и **set**.

Пример описания свойств:

```
public class Button: Control
{
   // закрытое поле, с которым связано свойство
   private string caption;
   // свойство
   public string Caption
   {
      get {                    // способ получения свойства
         return caption;
      }
      set {                      // способ установки свойства
         if (caption != value) {
            caption = value;
         }
      }
   }
   ...
}
```

Метод записи обычно содержит действия по проверке допустимости устанавливаемого значения, метод чтения может содержать, например, поддержку счетчика обращений к полю.

В программе свойство выглядит как поле класса, например:

```
Button ok = new Button();
ok.Caption = "OK";         // вызывается метод установки свойства
string s = ok.Caption;     // вызывается метод получения свойства
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод **get** должен содержать оператор **return**. В методе **set** используется параметр со стандартным именем **value**, который содержит устанавливаемое значение.

Добавим в класс **Monster**, описанный в листинге 2.9, свойства, позволяющие работать с закрытыми полями этого класса. Код класса несколько разрастется, зато упростится его использование.

Листинг 2.9. Класс Monster со свойствами

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      public Monster()
      {
         this.health = 100;
         this.ammo   = 100;
         this.name   = "Noname";
      }

      public Monster( string name ) : this()
      {
         this.name = name;
      }

      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }
      public int Health    // свойство Health связано с полем health
      {
         get
         {
            return health;
         }
         set
         {
            if (value > 0) health = value
            else           health = 0;
         }
      }

      public int Ammo      // свойство Ammo связано с п
```

```
      {
         get
         {
            return ammo;
         }
         set
         {
            if (value > 0) ammo = value;
            else           ammo = 0;
         }
      }

      public string Name    // свойство Name связано с полем name
      {
         get
         {
            return name;
         }
      }

      public void Passport()
      {
         Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
                           name, health, ammo);
      }

      string name;       // закрытые поля
      int health, ammo;
   }

   class Class1
   {
      static void Main()
      {
         Monster Masha = new Monster( 200, 200, "Masha" );
         Masha.Passport();
         --Masha.Health;              // использование свойств
         Masha.Ammo += 100;           // использование свойств
         Masha.Passport();
      }
   }
}
```

Результат работы программы:

```
Monster Masha    health = 200 ammo = 200
Monster Masha    health = 199 ammo = 300
```