САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ

- **Сотавов Абакар Капланович**
- **Ассистент кафедры Информатики**(наб. канала Грибоедова, 30/32, ауд. 2038
- e-mail: sotavov@unecon.ru

- Материалы на сайте: http://de.unecon.ru/course/view.php?id=440

Лекция 7. Классы: основные понятия

Основные элементы класса: поля, методы, конструкторы, свойства. Виды параметров методов.

Понятие объекта

- В реальном мире каждый предмет или процесс обладает свойствами и поведением (набором статических и динамических характеристик). *Поведение объекта* зависит от его *состояния* и *внешних воздействий*.
- Понятие объекта в программе совпадает с обыденным смыслом этого слова: *объект представляется как совокупность:*
  - *данных, характеризующих его состояние, и*
  - *функций их обработки, моделирующих его поведение*.
- Вызов функции на выполнение часто называют *посылкой сообщения* объекту.
- При создании объектно-ориентированной программы предметная область представляется в виде совокупности объектов. *Выполнение программы состоит в том, что объекты обмениваются сообщениями*.

Пример ОО-программы

```
class Двигатель
{   public void Запуск()
        {   Console.WriteLine( "пыщь пыщь" ); }
}

class Самолет
{   public Самолет()
    {   левый  = new Двигатель(); правый = new Двигатель(); }
    public void Запустить_двигатели()
    {   левый.Запуск();  правый.Запуск();    }
    Двигатель левый, правый;
}

class Client
{   static void Main()
    {   Самолет AH24_1 = new Самолет();
        AH24_1.Запустить_двигатели();
    }
}
```

Метод (функция) Запуск *класса* Двигатель

Конструктор класса Самолет

Метод Запустить_двигатели *класса* Самолет

посылка сообщения объекту правый

Поля класса Самолет

Объект AH24_1 *класса* Самолет

посылка сообщения объекту **AH24_1**

Результат работы программы:
пыщь пыщь
пыщь пыщь

Абстрагирование и инкапсуляция

При представлении реального объекта с помощью программного необходимо выделить в первом его существенные особенности и игнорировать несущественные. Это называется *абстрагированием*.

Таким образом, программный объект — это абстракция.

Детали реализации объекта скрыты, он используется через его *интерфейс* — совокупность правил доступа.

Скрытие деталей реализации называется *инкапсуляцией*. Это позволяет представить программу в укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объемом информации.

Итак, **объект — это инкапсулированная абстракция с четко определенным интерфейсом**.

Листинг: Иллюстрация инкапсуляции

```
class Двигатель
{   public void Запуск()
    {    Console.WriteLine( " пыщь пыщь " ); }
}
class Самолет
{    public Самолет()
     {    левый  = new Двигатель(); правый = new Двигатель(); }
     public void Запустить_двигатели()
     {   левый.Запуск();  правый.Запуск();   }
      Двигатель левый, правый ;              // скрытые поля данных
}
class Class1
{   static void Main()
     {   Самолет АН24_1 = new Самолет();
         АН24_1.Запустить_двигатели();
     }
}
```

Метод Main не знает, как именно запускаются двигатели

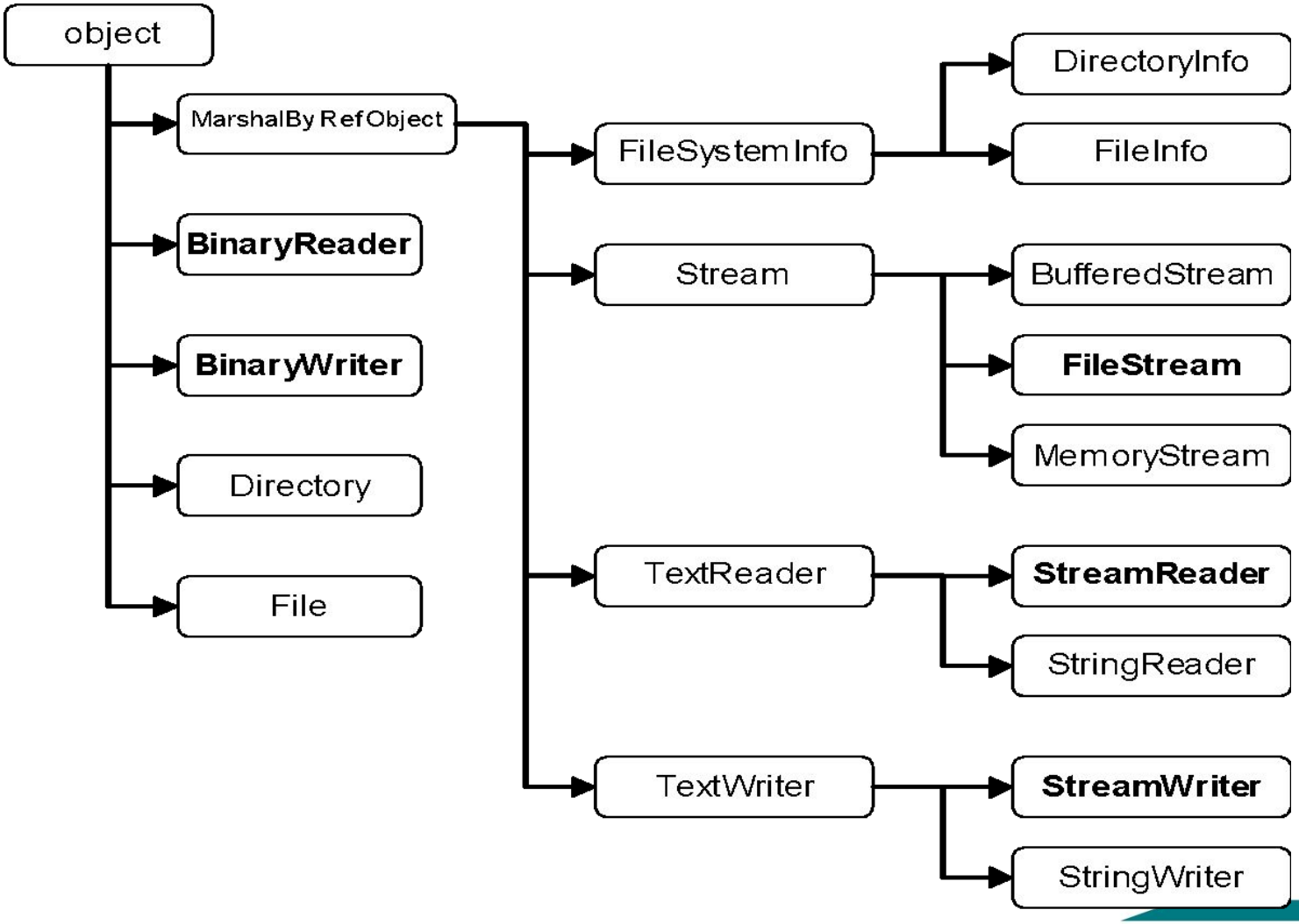
Результат работы программы:

```
вззззззззззз
вззззззззззз
вззззззззззз
вззззззззззз
```

Наследование

- Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого.
- Важное значение имеет возможность многократного использования кода. Для объекта можно определить наследников, корректирующих или дополняющих его поведение.
- *Наследование* применяется для:
  - исключения из программы повторяющихся фрагментов кода;
  - упрощения модификации программы;
  - упрощения создания новых программ на основе существующих.
- Благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.
- Наследование позволяет создавать *иерархии объектов*. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях.

Пример иерархии: классы .NET для работы с потоками



Пример полиморфизма

Ëèñòèíã.zip

Достоинства ООП

- использование при программировании понятий, близких к предметной области;
- возможность успешно управлять большими объемами исходного кода благодаря инкапсуляции, то есть скрытию деталей реализации объектов и упрощению структуры программы;
- возможность многократного использования кода за счет наследования;
- сравнительно простая возможность модификации программ;
- возможность создания и использования библиотек объектов.

Недостатки ООП

- идеи ООП не просты для понимания и в особенности для практического использования
- для эффективного использования существующих объектно-ориентированных систем и библиотек требуется **большой объем первоначальных знаний**
- **неграмотное применение ООП может привести к значительному ухудшению характеристик разрабатываемой программы**
- некоторое снижение быстродействия программы, связанное с использованием виртуальных методов

1. В предметной области выделяются понятия, которые можно использовать как классы.
   Кроме классов из предметной области, появляются классы, связанные с реализацией.
2. Определяются операции над классами, которые впоследствии станут методами класса. Их можно разбить на группы:
   - связанные с конструированием и копированием объектов
   - для поддержки связей между классами, которые существуют в прикладной области
   - позволяющие представить работу с объектами в удобном виде.
3. Определяются методы, которые будут виртуальными.
4. Определяются зависимости между классами.

Процесс создания иерархии классов - итерационный. Например, можно в двух классах выделить общую часть в базовый класс и сделать их производными.

Понятие класса (повторение)

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 (*методы*).

Все классы .NET имеют общего предка — класс object, и организованы в единую иерархическую структуру.

Классы логически сгруппированы в пространства имен, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными.

Любая программа использует пространство имен System.

Описание класса

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

Имя класса задается по общим правилам.

Тело класса — список описаний его элементов, заключенный в фигурные скобки.

Атрибуты задают дополнительную информацию о классе.

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы.

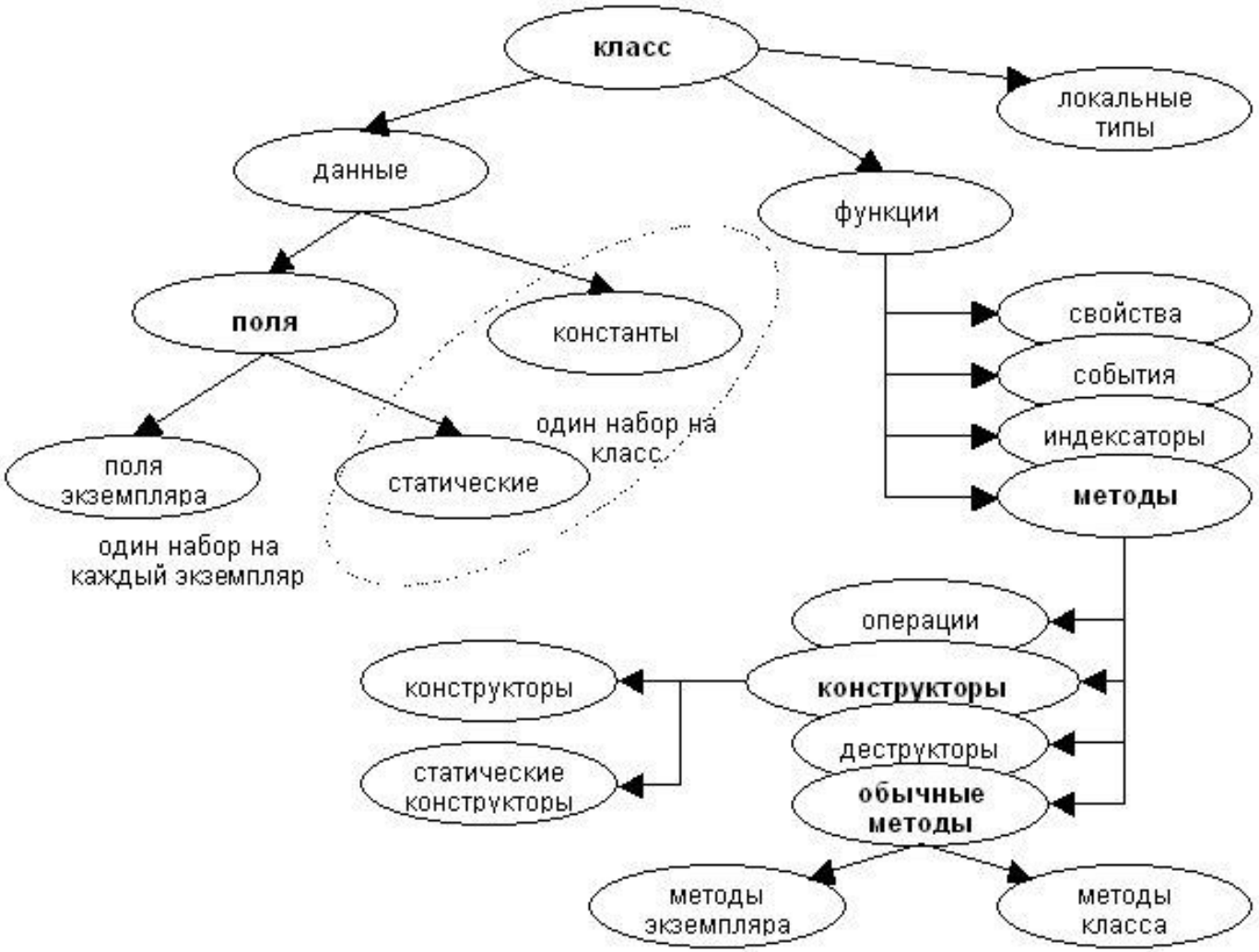
Простейшие примеры описания класса:

```
class Demo {}                          // пустой класс
public class Двигатель                 // класс с одним методом
{   public void Запуск()
    {    Console.WriteLine( " пыщь пыщь " ); }
}
```

Спецификаторы класса

Спецификатор	Описание
new	(для вложенных классов). Задает новое описание класса взамен унаследованного от предка. Применяется в иерархиях
public	Доступ не ограничен
protected	Используется для вложенных классов. Доступ только из элементов данного и производных классов
internal	Доступ только из данной программы (сборки)
protected internal	Доступ только из данного и производных классов или из данной программы (сборки)
private	(для вложенных классов). Доступ только из элементов класса, внутри которого описан данный класс
abstract	Абстрактный класс. Применяется в иерархиях
sealed	Бесплодный класс. Применяется в иерархиях
static	Статический класс.

Элементы класса



Описание объекта (экземпляра)

Класс является обобщенным понятием, определяющим характеристики и поведение множества конкретных объектов этого класса, называемых экземплярами (объектами) класса.

Объекты создаются явным или неявным образом (либо программистом, либо системой). Программист создает экземпляр класса с помощью операции new:

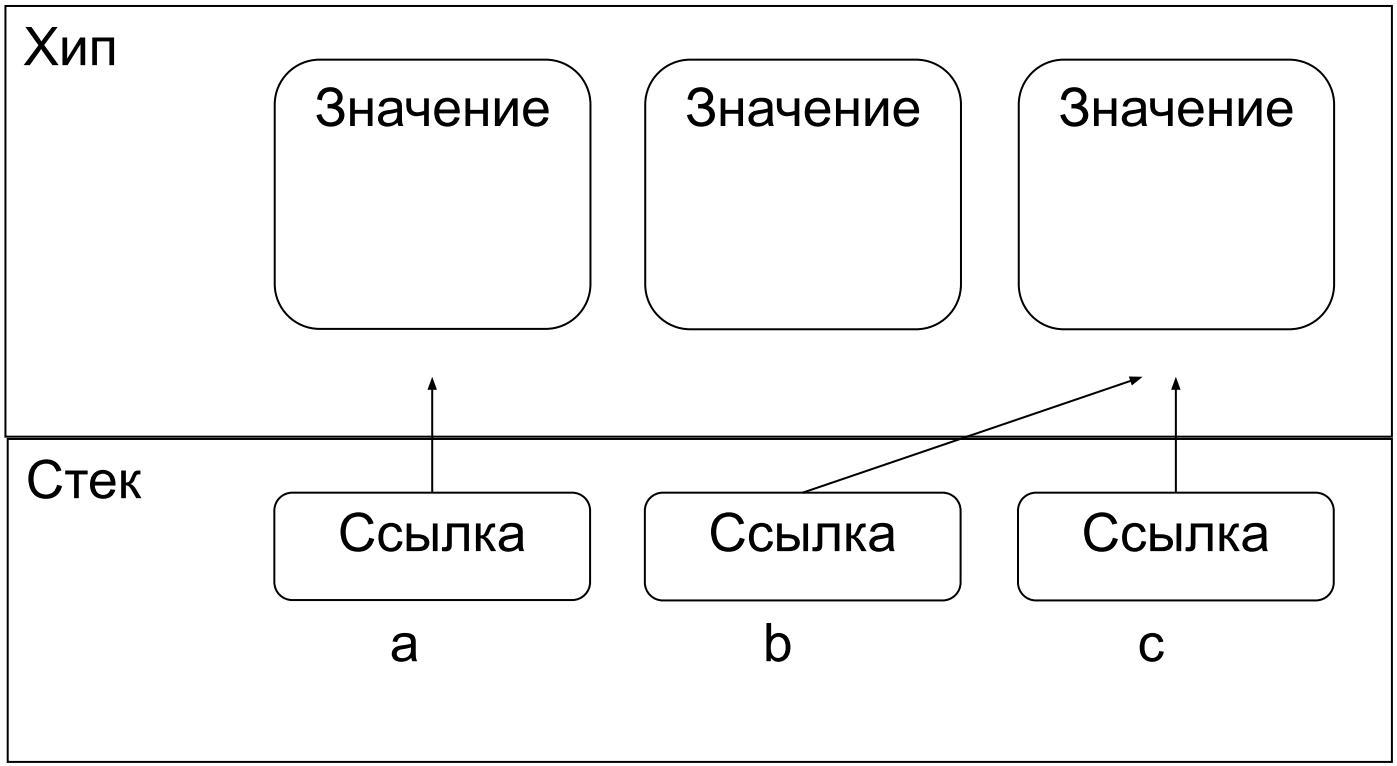
```
Demo a = new Demo();

Monster Vasia = new Monster();

Monster Petya = new Monster("Петя");
```

Для каждого объекта при его создании в памяти выделяется отдельная область для хранения его данных.

Кроме того, в классе могут присутствовать статические элементы, которые существуют в единственном экземпляре для всех объектов класса.

Функциональные элементы класса всегда хранятся в единственном экземпляре.

Присваивание и сравнение объектов (повторение)



Данные: поля и константы

Данные, содержащиеся в классе, могут быть переменными или константами.

Переменные, описанные в классе, называются **полями** класса.

При описании полей можно указывать атрибуты и спецификаторы, задающие различные характеристики элементов:

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

```
public int a = 1;

public static string s = "Demo";

double y;
```

Все поля сначала автоматически инициализируются нулем соответствующего типа (например, полям типа int присваивается 0, а ссылкам на объекты — значение null). После этого полю присваивается значение, заданное при его явной инициализации.

Спецификаторы полей и констант класса

Спецификатор	Описание
new	Новое описание поля, скрывающее унаследованный элемент класса
public	Доступ к элементу не ограничен
protected	Доступ только из данного и производных классов
internal	Доступ только из данной сборки
protected internal	Доступ только из данного и производных классов и из данной сборки
private	Доступ только из данного класса
static	Одно поле для всех экземпляров класса
readonly	Поле доступно только для чтения
volatile	Поле может изменяться другим процессом или системой

Пример класса

```
using System;
namespace CA1
{   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
  }}}
```

Методы

Метод — функциональный элемент класса, реализующий вычисления или другие действия. Методы определяют поведение класса и составляют его **интерфейс**.

Метод — 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.

```
double a = 0.1;
double b = Math.Sin(a);
double c = Math.Sin(b-2*a);
```

```
Console.WriteLine(a);
```

Синтаксис метода

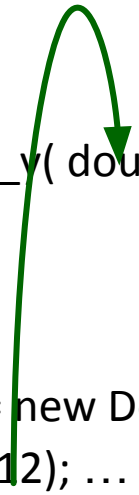
[атрибуты] [спецификаторы] тип имя_метода **([параметры])** тело_метода

Спецификаторы: new, **public**, protected, internal, protected internal, private, static, virtual, sealed, override, abstract, extern.

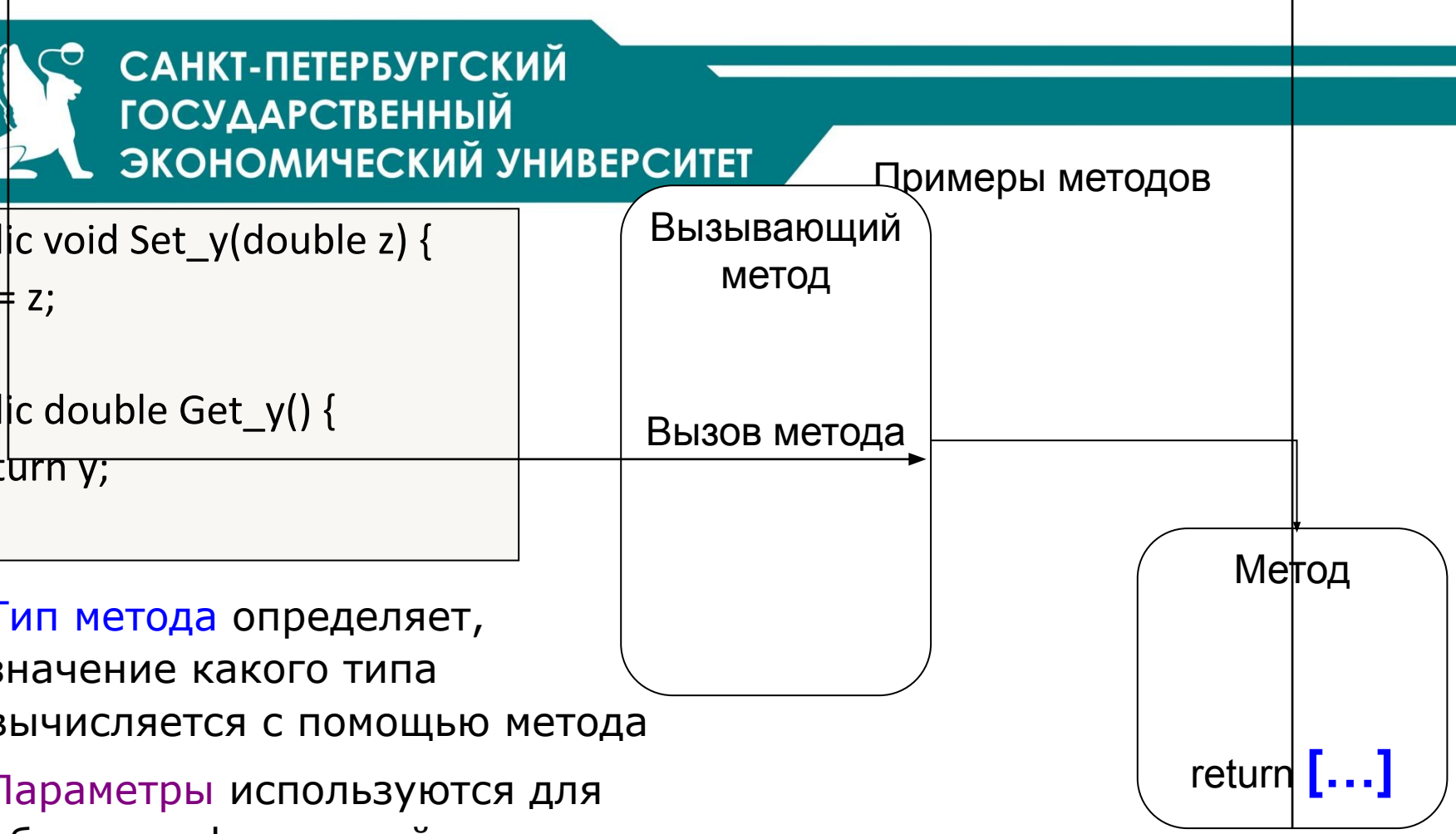
Метод класса имеет непосредственный доступ к его полям.

Пример:

```
class Demo
{  double y;                          // закрытое поле класса

   public void Set_y( double z )      // открытый метод класса
   {    y = z;
   }
}
… Demo demo = new Demo();             // где-то в методе другого класса
   demo.Set_y(3.12); …                // вызов метода
```



```
public void Set_y(double z) {
    y = z;
}
public double Get_y() {
   return y;
}
```



- Тип метода определяет, значение какого типа вычисляется с помощью метода
- Параметры используются для обмена информацией с методом. *Параметр - локальная переменная, которая при вызове метода принимает значение соответствующего* ***аргумента***.

Параметры методов

Параметры 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, имя и, возможно, вид параметра.

Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой сигнатуру метода — то, по чему один метод отличают от других.

В классе не должно быть методов с одинаковыми сигнатурами.

Метод, описанный со спецификатором static, должен обращаться только к статическим полям класса.

Статический метод вызывается через имя класса, а обычный — через имя экземпляра.

Вызов метода

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода.
3. Каждому из параметров сопоставляется соответствующий аргумент. При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.
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.

Примеры передачи аргументов в метод

```
class Class1
{   static int Max(int a, int b)     // метод выбора макс. значения
    {
        return a > b ? a : 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 );       // аргументы совместимого типа
        Console.WriteLine( y );              // результат: 4
        int z = Max(a + t1, t1 / 2 * b);   // аргументы-выражения
        Console.WriteLine( z );              // результат: 5
}}
```

Аргументы передаются:

- По значению
- По адресу (ссылке)

При передаче по значению метод получает копии значений аргументов, и операторы метода работают с этими копиями.

При передаче по ссылке (*по адресу*) метод получает копии адресов аргументов и осуществляет доступ к аргументам по этим адресам.

Типы параметров

В C# четыре типа параметров:

параметры-значения - для исходных данных метода;

параметры-ссылки (**ref**) - для изменения аргумента;

выходные параметры (**out**) – для формирования аргумента;

параметры-массивы (**params**) – для переменного кол-ва аргументов.

по адресу

Пример:

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

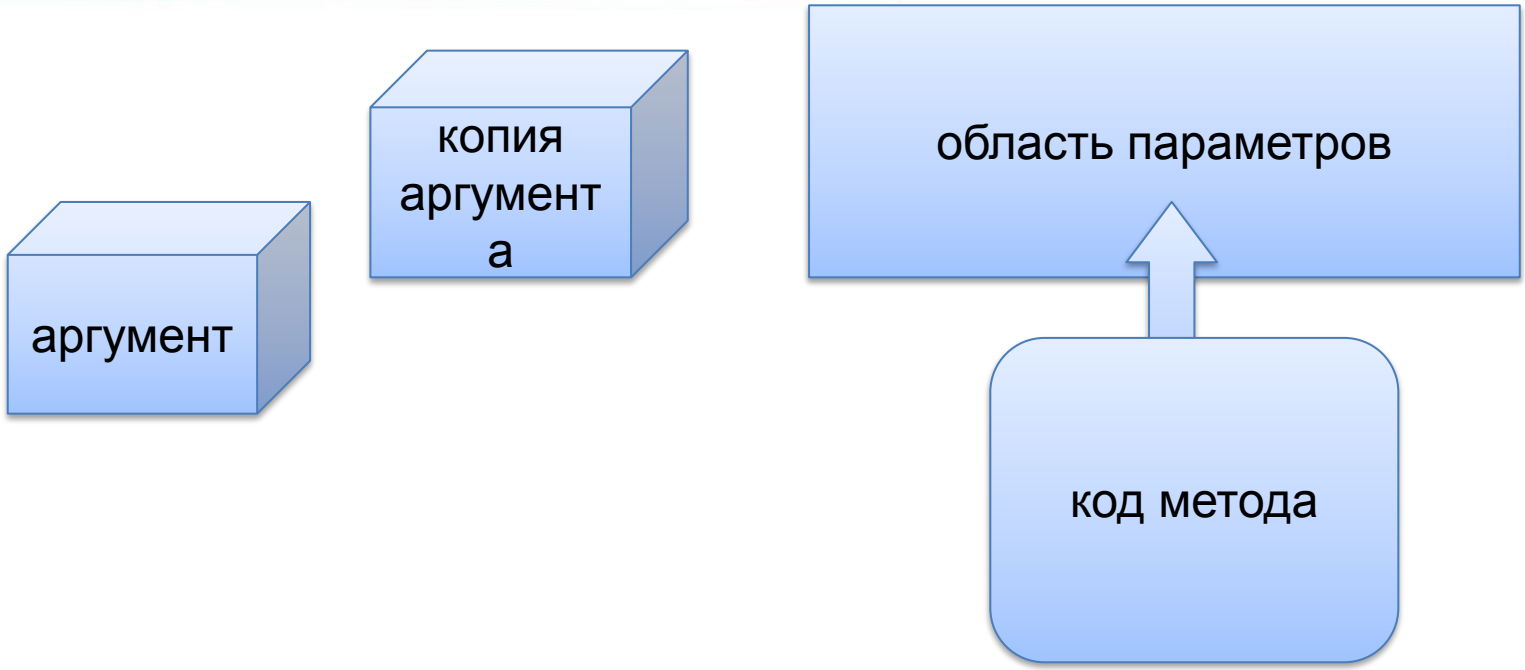
параметр-значение

параметр-ссылка

выходной параметр

параметр-массив

Передача аргумента по значению

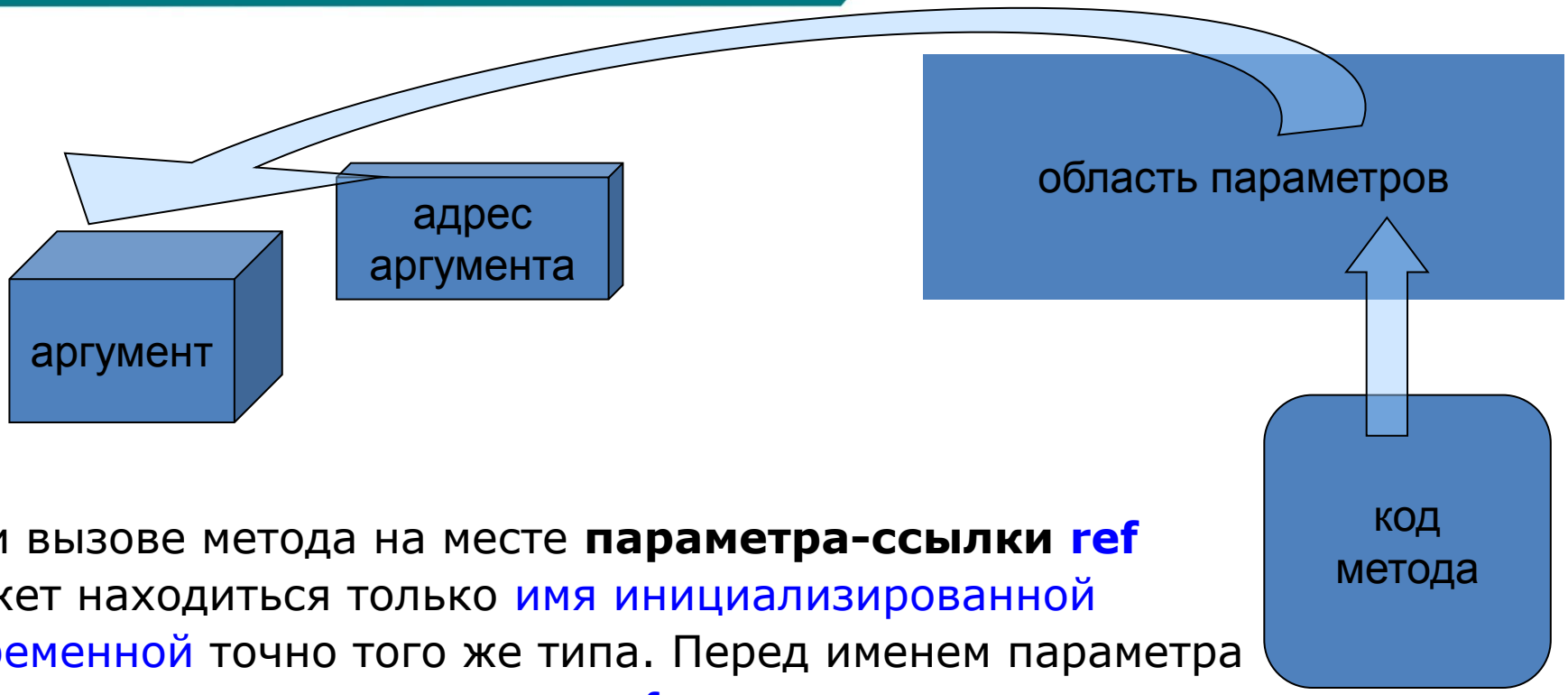


- При вызове метода на месте параметра, передаваемого по значению, может находиться выражение (а также его частные случаи — переменная или константа).
- Должно существовать неявное преобразование типа выражения к типу параметра.

Пример: счетчик

```
class Counter
{   public void Inc(int delta)
    {
        n += delta;
     }
      public override string ToString() { return n.ToString(); }
      int n;
}
class Program
{   static void Main(string[] args)
    {   Counter num = new Counter();
        num.Inc(4);
        int a = 3;
        num.Inc(2*a);
        Console.WriteLine("значение счетчика " + num);
}}
```

Передача аргумента по ссылке (ref, out)



- При вызове метода на месте **параметра-ссылки ref** может находиться только имя инициализированной переменной точно того же типа. Перед именем параметра указывается ключевое слово ref.
- При вызове метода на месте **выходного параметра out** может находиться только имя переменной точно того же типа. Ее инициализация не требуется. Перед именем параметра указывается ключевое слово out.

Пример: параметры-значения и ссылки ref

```
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 );
}}}
```

Результат работы программы:
до вызова 2 4
внутри метода 44 33
после вызова 2 33

Пример: выходные параметры out

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void P( int x, out int y )
        {
            x = 44; y = 33;
            Console.WriteLine( "внутри метода {0} {1}", x, y );
        }
        static void Main()
        {
            int a = 2, b;        // инициализация b не требуется

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

Результат работы программы:

внутри метода 44 33
после вызова 2 33

Summary: Правила применения параметров

Для параметров-значений используется передача по значению. Этот способ применяется для исходных данных метода.

При вызове метода на месте аргумента параметра-значения может быть выражение (а также его частные случаи — переменная или константа). Должно существовать неявное преобразование типа выражения к типу параметра.

Параметры-ссылки и выходные параметры передаются по адресу. Этот способ применяется для передачи побочных результатов метода.

При вызове метода на месте аргумента параметра-ссылки **ref** может находиться только имя инициализированной переменной точно того же типа. Перед именем параметра и аргумента указывается ключевое слово ref.

При вызове метода на месте выходного параметра **out** может находиться только имя переменной точно того же типа. Ее инициализация не требуется. Перед именем параметра и аргумента указывается ключевое слово out.

Методы с переменным количеством аргументов

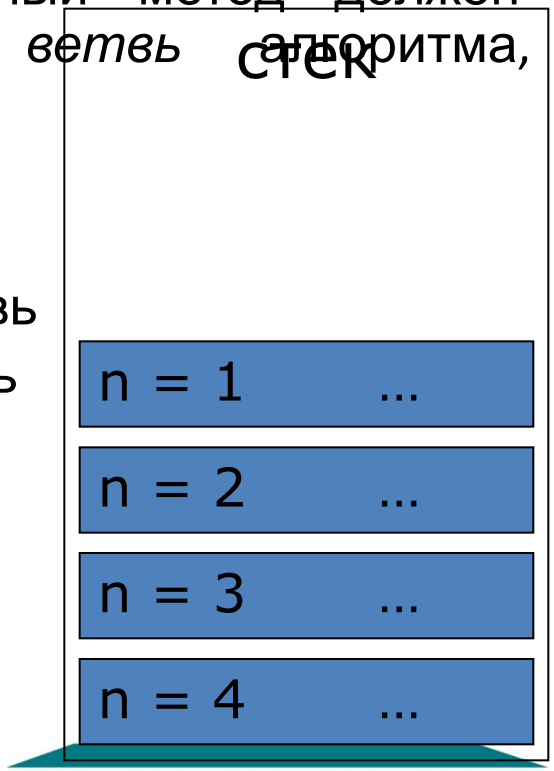
```
class Class1
{    public static double Average( params int[] a )
     {  if ( a.Length == 0 )
                throw new Exception( "Недостаточно аргументов");
        double sum = 0;
        foreach ( int elem in a ) sum += elem;
        return sum / a.Length;
      }
     static void Main()
     {   try
         {  short z = 1, e = 13, w = 4;
            Console.WriteLine( Average( z, e, w ) );        //  6
            byte v = 18;
            Console.WriteLine( Average( z, e, w, v) );      //  9
            int[] b = { -11, -4, 12, 14, 32, -1, 28 };
            Console.WriteLine( Average( b ) );              // 38
            Console.WriteLine( Average() );   // Недостаточно аргументов
         }
         catch( Exception e ) { Console.WriteLine( e.Message ); return; }
         catch                { Console.WriteLine( "ой-кшмр" ); return; }
   }}
```

Рекурсивные методы

Рекурсивным называется метод, который вызывает сам себя (*прямая рекурсия*). *Косвенная* рекурсия - когда два или более метода вызывают друг друга.

Для завершения вычислений каждый рекурсивный метод должен содержать *хотя бы одну нерекурсивную ветвь* алгоритма, заканчивающуюся оператором возврата.

```
long fact( long n ) {
   if ( n == 0 || n == 1 ) return 1;       // нерекурсивная ветвь
   return ( n * fact( n – 1 ) );             // рекурсивная ветвь
}
… long m = fact(4);

// или:
long fact( long n ) { return ( n > 1 ) ? n * fact( n – 1 ) : 1; }
```



Характеристики рекурсии

Достоинство рекурсии: компактность записи.

Недостатки: опасность переполнения стека; расход времени и памяти на повторные вызовы метода и передачу ему копий параметров.

Рекурсивные методы используются в основном для работы с рекурсивными структурами данных, например, бинарными деревьями.

Ключевое слово this

Чтобы обеспечить работу метода с полями того объекта, для которого он был вызван, в метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию объект.

В явном виде параметр this применяется:

1) чтобы возвратить из метода ссылку на вызвавший объект:

```
class Demo
{     double y;
      public Demo T()    { return this; }
// 2) для идентификации поля, если его имя совпадает с
//     именем параметра метода:
      public void Set_y( double y ) { this.y = y; }
}
```

Конструкторы

Конструктор – особый вид метода, предназначенный для инициализации

объекта (конструктор экземпляра) или

класса (статический конструктор).

Конструктор экземпляра инициализирует данные экземпляра, конструктор класса — данные класса.

Конструкторы экземпляра

Конструктор вызывается автоматически при создании объекта класса с помощью операции **new**. Имя конструктора совпадает с именем класса.

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**.

Пример класса с конструктором

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

   class Class1
   {   static void Main()
      {
         Demo a = new Demo( 300, 0.002 );     // вызов конструктора
         Demo b = new Demo( 1, 5.71 );        // вызов конструктора
         …
    } }
```

Пример класса с двумя конструкторами

```
class Demo
   {
      public Demo( int a )              // конструктор 1
      {
         this.a = a;
         this.y = 0.002;
      }
      public Demo( double y )           // конструктор 2
      {
         this.a = 1;
         this.y = y;
      }
      ...
   }
      ...
         Demo x = new Demo( 300 );      // вызов конструктора 1
         Demo y = new Demo( 5.71 );     // вызов конструктора 2
```

Статические конструкторы

Статический конструктор (конструктор класса) инициализирует статические поля класса.

Он не имеет параметров, его нельзя вызвать явным образом. Система сама определяет момент, в который требуется его выполнить. Гарантируется, что это происходит до создания первого экземпляра объекта и до вызова любого статического метода.

Статический конструктор, как и любой статический метод, не имеет доступа к полям экземпляра.

Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство определяет методы доступа к закрытому полю.

Свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом.

Синтаксис свойства:

[спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

При обращении к свойству автоматически вызываются указанные в нем блоки чтения (**get)** и установки (**set)**.

Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует get - только для записи (write-only).

Пример: счетчик (свойства)

```
class Counter
{     public Counter( int n = 0 )
            { this.n = n > 0 ? n : 0; }
      public int N
      {  get { return n; }
         set { n = value > 0 ? value : 0; }
      }
// или: set { if (value > 0) n = value; else throw new Exception();}
      int n;       // поле, связанное со свойством N
}
class Program
{    static void Main(string[] args)
     {    Counter num = new Counter();
          num.N = 5;          //   работает set
          int a = num.N;      //   работает get
          num.N++;           //   работает get, а потом set
          ++num.N;           //   работает get, а потом set
}}
```

Перегрузка методов

Перегрузкой методов называется использование нескольких методов с одним и тем же именем, но различными типами параметров.

Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Это называется ***разрешением*** (resolution) перегрузки.

```
        // Возвращает наибольшее из двух целых:
int max( int a, int b )
        // Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
        // Возвращает наибольшее из первого параметра
        // и длины второго:
int max ( int a, string b )
...
Console.WriteLine( max( 1, 2 ) );
Console.WriteLine( max( 1, 2, 3 ) );
Console.WriteLine( max( 1, "2" ) );
```

Перегрузка методов является проявлением *полиморфизма*

Пример: счетчик (перегрузка)

```
class Counter
{     public Counter() { }
      public Counter( int n ) { this.n = n > 0 ? n : 0;  }

      public void Inc() { ++n; }
      public void Inc( int delta ){ n += Math.Abs(delta); }
      int n;
      ...
}
class Program
{    static void Main(string[] args)
     {    Counter num1 = new Counter();
          Counter num2 = new Counter(128);
          num1.Inc(4);
          num1.Inc();
          ...
     }
}
```

В C# можно переопределить для своих классов действие большинства операций. Это позволяет применять экземпляры объектов в составе выражений аналогично переменным стандартных типов:

```
MyObject a, b, c;  ...
c = a + b;              // операция сложения класса MyObject
```

Определение собственных операций класса называют ***перегрузкой операций***.

Операции класса описываются с помощью методов специального вида (***функций-операций***):

public static имя_класса operator операция(параметры) {… }

Пример: public static MyObject operator --(MyObject m) { … }

В C# три вида операций класса: унарные, бинарные и операции преобразования типа.

операция должна быть описана как открытый статический метод класса (спецификаторы **public static**);

параметры в операцию должны передаваться **по значению** (то есть не должны предваряться ключевыми словами ref или out);

сигнатуры всех операций класса должны различаться;

типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

Пример: счетчик (операция ++)

```
class Counter
{     public Counter( int n = 0 ) { this.n = n > 0 ? n : 0; }
      public static Counter operator ++(Counter param)
      {
         Counter temp = new Counter(param.n + 1);
         return temp;
      }
      int n;
}
class Program
{    static void Main(string[] args)
     {    Counter num = new Counter();
          num++;    ++num;
          ...
     }
}
```

Правила оформления унарной операции класса

Параметр функции-операции должен иметь тип этого класса.

Операция должна возвращать:

для операций +, -, ! и ~ величину любого типа;

для операций ++ и -- величину типа класса, для которого она определяется.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Префиксный и постфиксный инкремент/декремент не различаются

Бинарные операции

Можно определять:

+ - * / % & | ^ << >> == != > < >= <=

Примеры заголовков бинарных операций:

```
public static MyObject operator +  ( MyObject m1, MyObject m2 )
public static bool     operator == ( MyObject m1, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение.

Пример бинарных операций класса

```
class Monster {
  public static Monster operator +( Monster m, int k )
  { Monster temp = new Monster(m.health+k, m.ammo, m.name);
    return temp;
  }
 public static Monster operator +( int k, Monster m )
  { Monster temp = new Monster(m.health+k, m.ammo, m.name);
    return temp;
  }
  …
}
 …
Monster vasia = new Monster();
Monster masha = vasia + 10;
Monster petya = 5 + masha;
…
```

Пример: счетчик (операция +)

```
class Counter
{    ...
      public static Counter operator +(Counter param, int delta)
     {  Counter temp = new Counter(param.n + delta);
        return temp;
     }
     public static Counter operator +(int delta, Counter param)
     {  Counter temp = new Counter(param.n + delta);
        return temp;
     }
     int n;
}
class Program
{    static void Main(string[] args)
    {    Counter num = new Counter(); Counter num2 = new Counter();
         num2 = num + 3;      num2 = 3 + num;
          ...
     }
}
```

Операции преобразования типа

Обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных.

implicit operator тип (параметр) // неявное преобразование

explicit operator тип (параметр) // явное преобразование

Выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразование либо типа класса к другому типу, либо наоборот.

```
public static implicit operator int( Monster m )
{
   return m.health;
}
public static explicit operator Monster( int h )
{
   return new Monster( h, 100, "FromInt" );
} ...
Monster Masha = new Monster( 200, 200, "Masha" );
int i = Masha;                  // неявное преобразование
Masha = (Monster) 500;      // явное преобразование
```

Пример: счетчик (приведение типа)

```
class Counter
{    ...
      public static implicit operator int(Counter param)
     {   return param.n;   }
     public static implicit operator Counter(int n)
     {   return new Counter(n);  }
     int n;
}
class Program
{    static void Main(string[] args)
    {    Counter num = new Counter();
         int a = num;
         num = 4;
         a = num + 3;
;
          ...
     }
}
```

Неявное преобразование выполняется автоматически:

- при присваивании объекта переменной целевого типа;
- при использовании объекта в выражении, содержащем переменные целевого типа;
- при передаче объекта в метод на место параметра целевого типа;
- при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа, например:

```
int a = (int) имя_объекта;
```

Диструкторы

Синтаксис деструктора:
[атрибуты] [extern] ~имя_класса() тело

Вложенные типы

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

СПАСИБО ЗА ВНИМАНИЕ!

ИНЖЭКОН ENGECON

ФИНЭК
Санкт-Петербург

СПб ГУСЭ

ОБЪЕДИНЯЯ ЛУЧШЕЕ