

Университет машиностроения

Кафедра «Автоматика и процессы управления»

Дисциплина

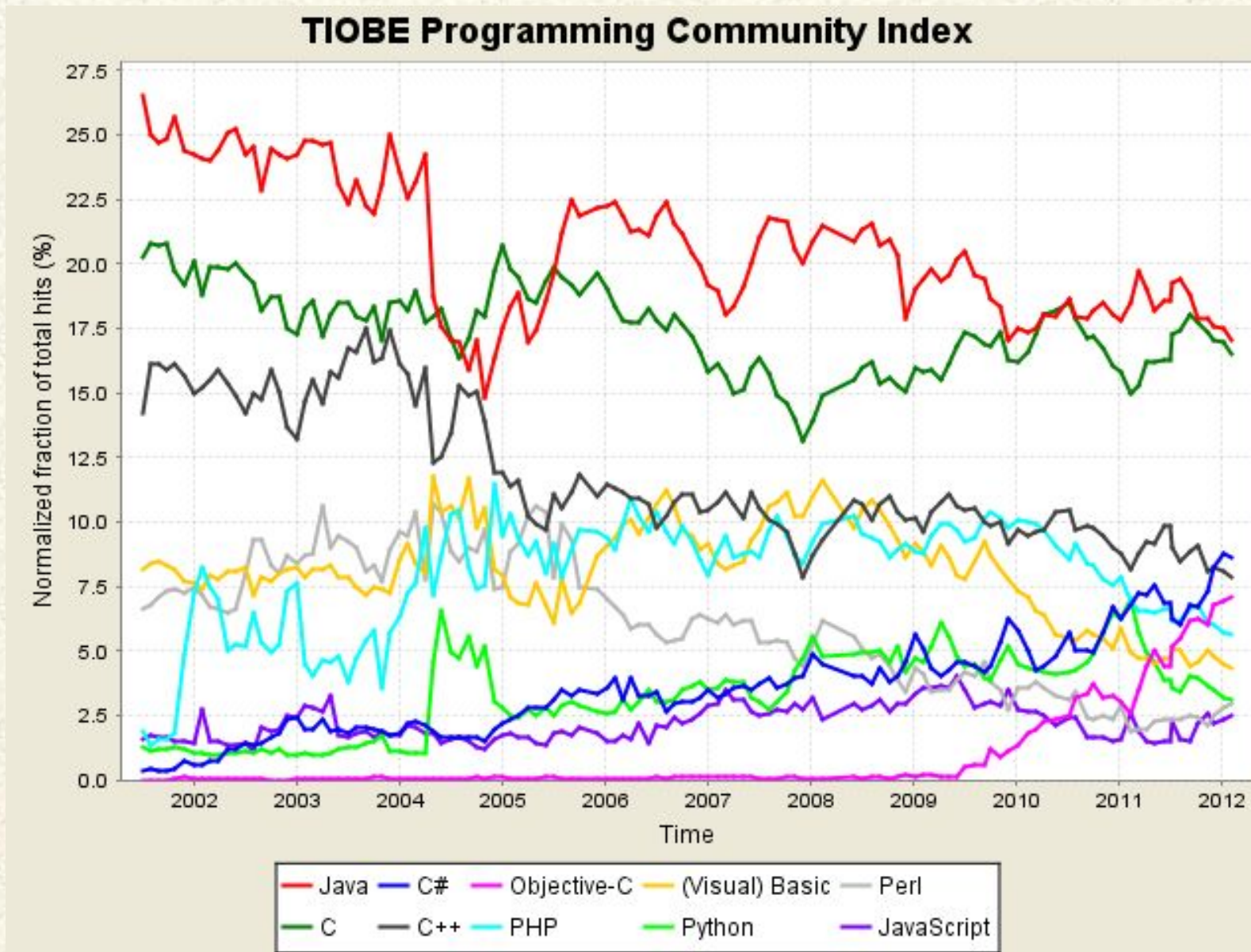
Информационные технологии

2 семестр

Тема 16

**Объектно-ориентированное
программирование на языке C#**

Рейтинг языков программирования на февраль 2012 г.



Языки ООП:

Java

C#

C++

Objective-C

Python

JavaScript

Языки с
поддержкой
ООП:

PHP

Visual Basic

Perl

Языки без
поддержки
ООП:

C

UML Unified Modeling Language — унифицированный язык моделирования

Язык для спецификации, визуализации, конструирования и документирования программных продуктов.

Также используется в бизнес-моделировании и моделировании любых иных (не программных) систем.

UML позволяет описывать следующие аспекты модели системы:

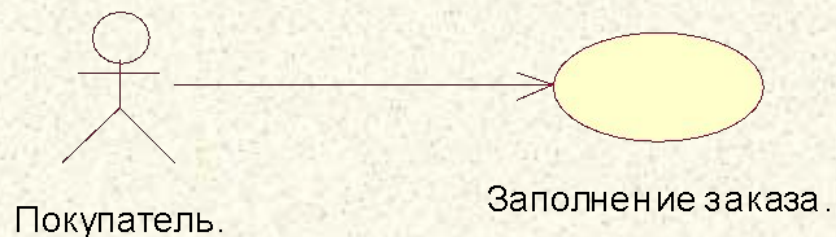
Диаграммы вариантов использования (use case diagrams)

Диаграммы классов (class diagrams)

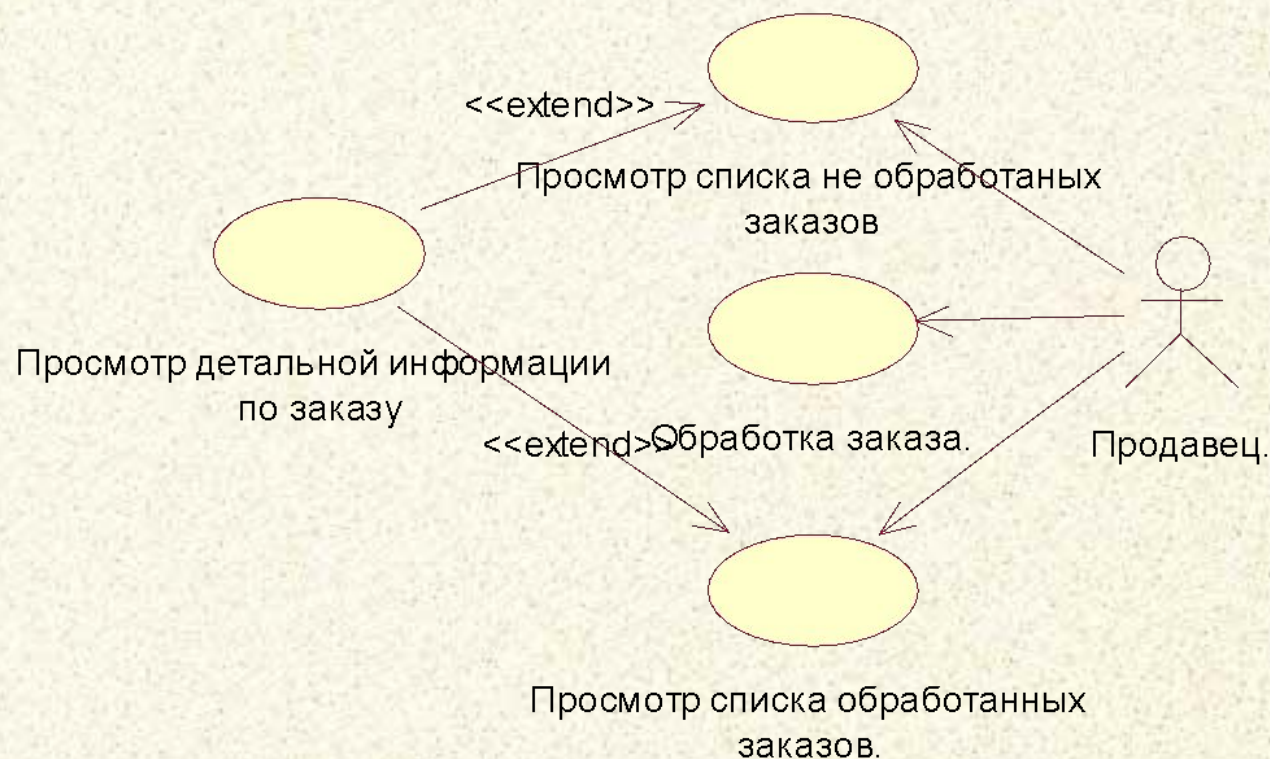
Диаграммы поведения

- Диаграммы состояний (statechart diagrams)
- Диаграммы действий (activity diagrams)
- Диаграммы взаимодействия (interaction diagrams)
 - Диаграммы последовательностей (sequence diagrams)
 - Диаграммы взаимодействий (collaboration diagrams)
- Диаграммы реализации (implementation diagrams)
 - Диаграммы компонент (component diagram)
 - Диаграммы развертывания (deployment diagram)

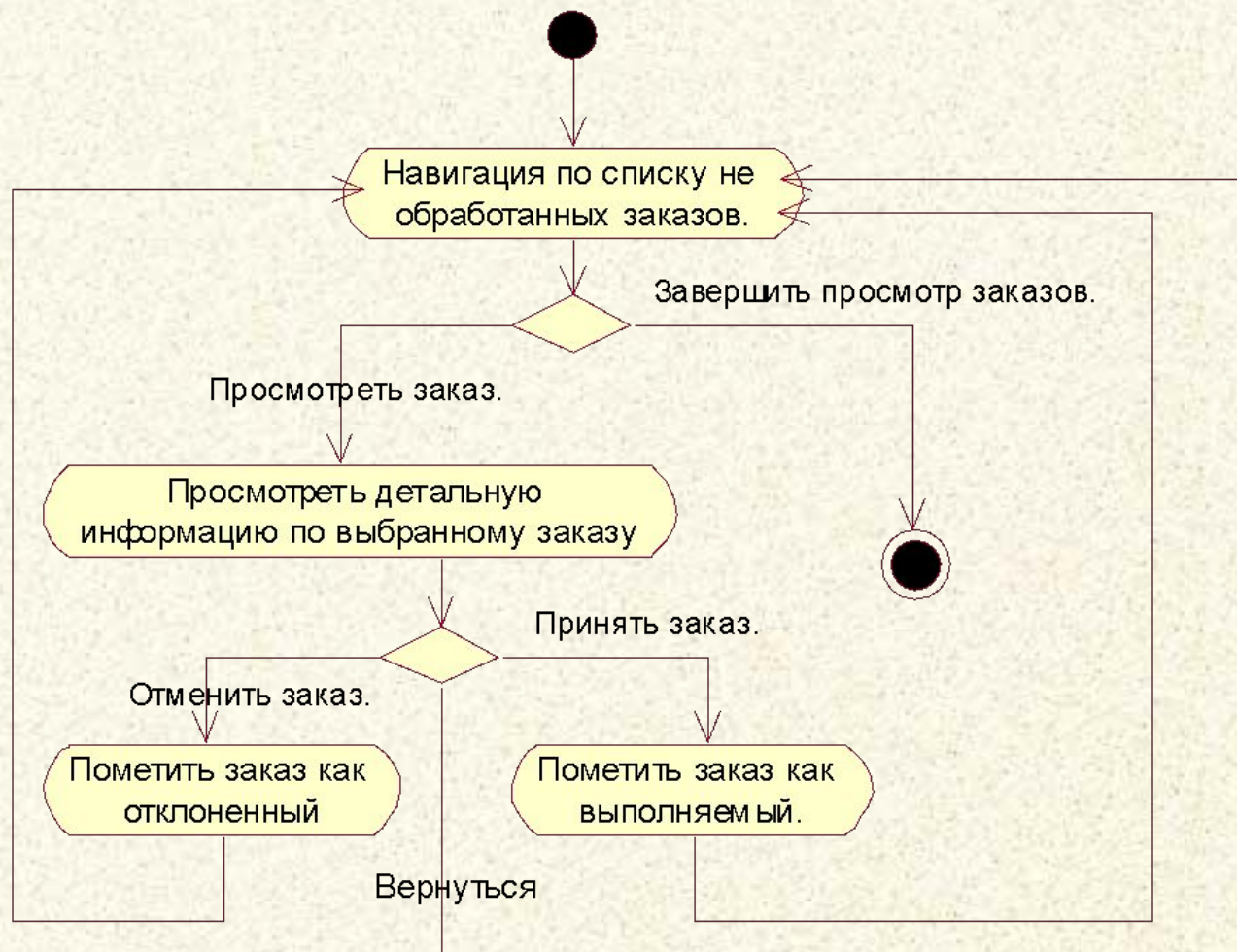
Use case diagrams — Диаграммы вариантов использования



**Проектируется:
Объекты и
методы**



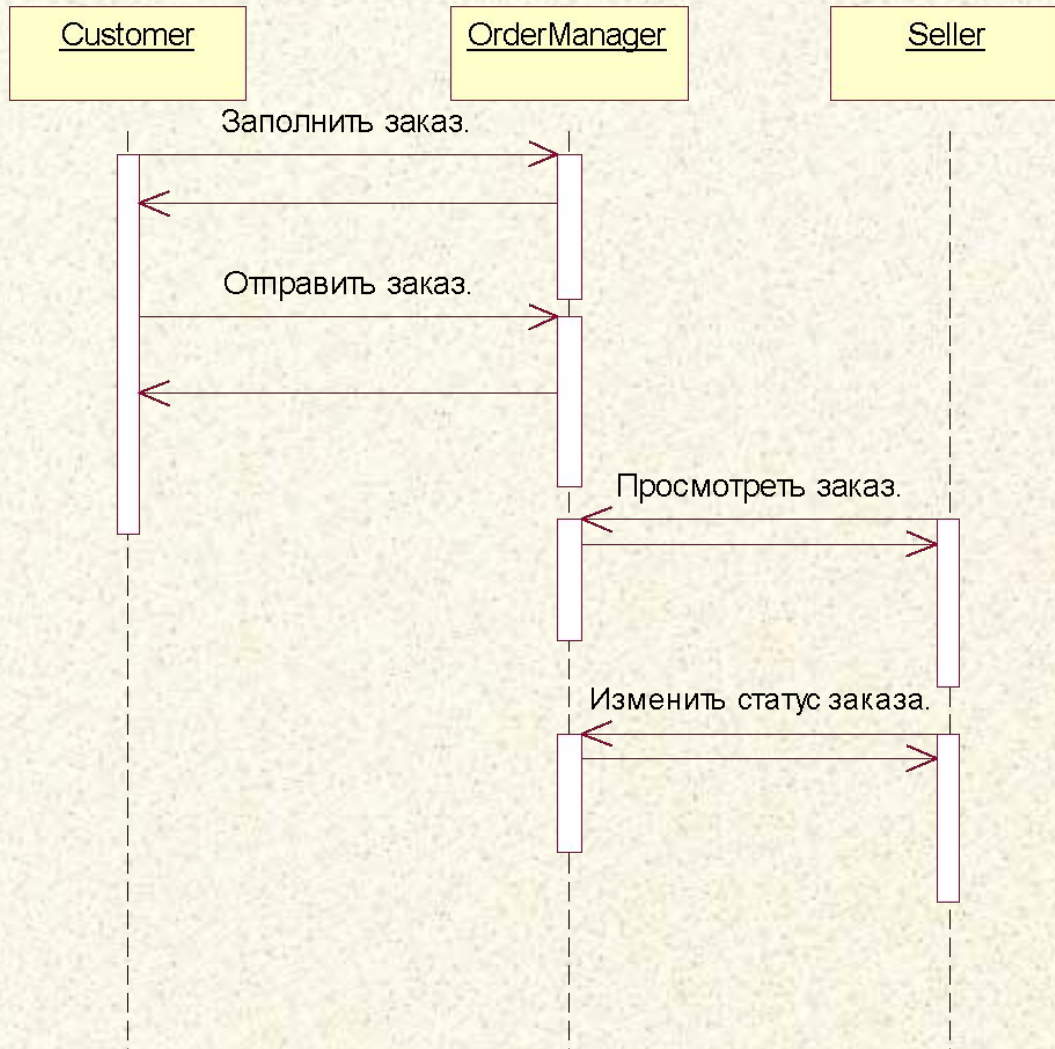
Activity diagrams — Диаграммы деятельности



Проектируется:
Реализация
методов

Sequence diagrams —

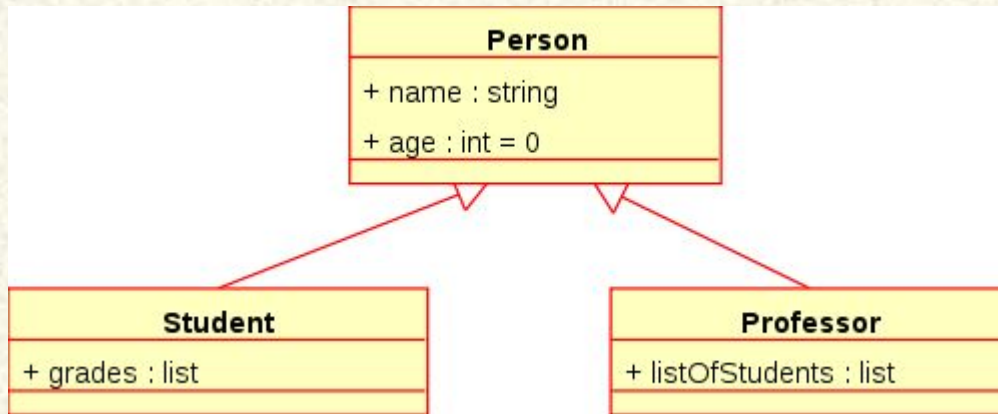
Диаграммы последовательностей действий



Проектируется:
Взаимодействие объектов

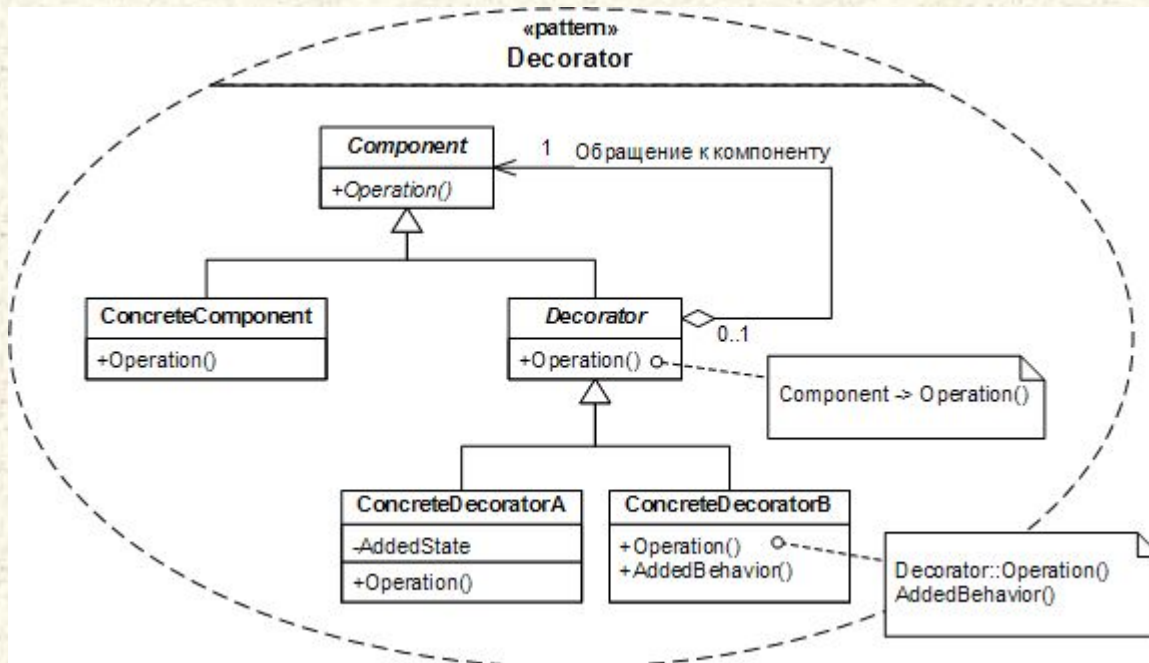
Class diagrams — Диаграммы классов

Generalization — Обобщение



Проектируется:
Структура и наследование объектов

Composite structure diagram — Диаграмма композитной (составной) структуры



Проектируется:
Структура классов и взаимодействие их элементов

Понятие объекта

- В реальном мире каждый предмет или процесс обладает набором статических и динамических характеристик (свойствами и поведением). *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.

Абстракция 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.

Наследование

- ❑ 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.

Полиморфизм

- ❑ ООП позволяет писать гибкие, расширяемые и читабельные программы.
- ❑ Во многом это обеспечивается благодаря **полиморфизму**, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа.
- ❑ Полиморфизм чаще всего реализуется через механизм виртуальных методов.

Реализация виртуального метода может быть определена во время исполнения программы. Конкретную реализацию виртуального метода нужно определить в классе-наследнике, с помощью которого и будет проводиться создание объекта.

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

- ❑ некоторое снижение быстродействия программы, связанное с использованием виртуальных методов;
- ❑ идеи ООП не просты для понимания и в особенности для практического использования;
- ❑ для эффективного использования существующих объектно-ориентированных систем требуется большой объем первоначальных знаний;
- ❑ неправильное проектирование значительно усложняет характеристики разрабатываемой программы.

Неграмотное применение ООП может привести к значительному ухудшению характеристик разрабатываемой программы!

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.

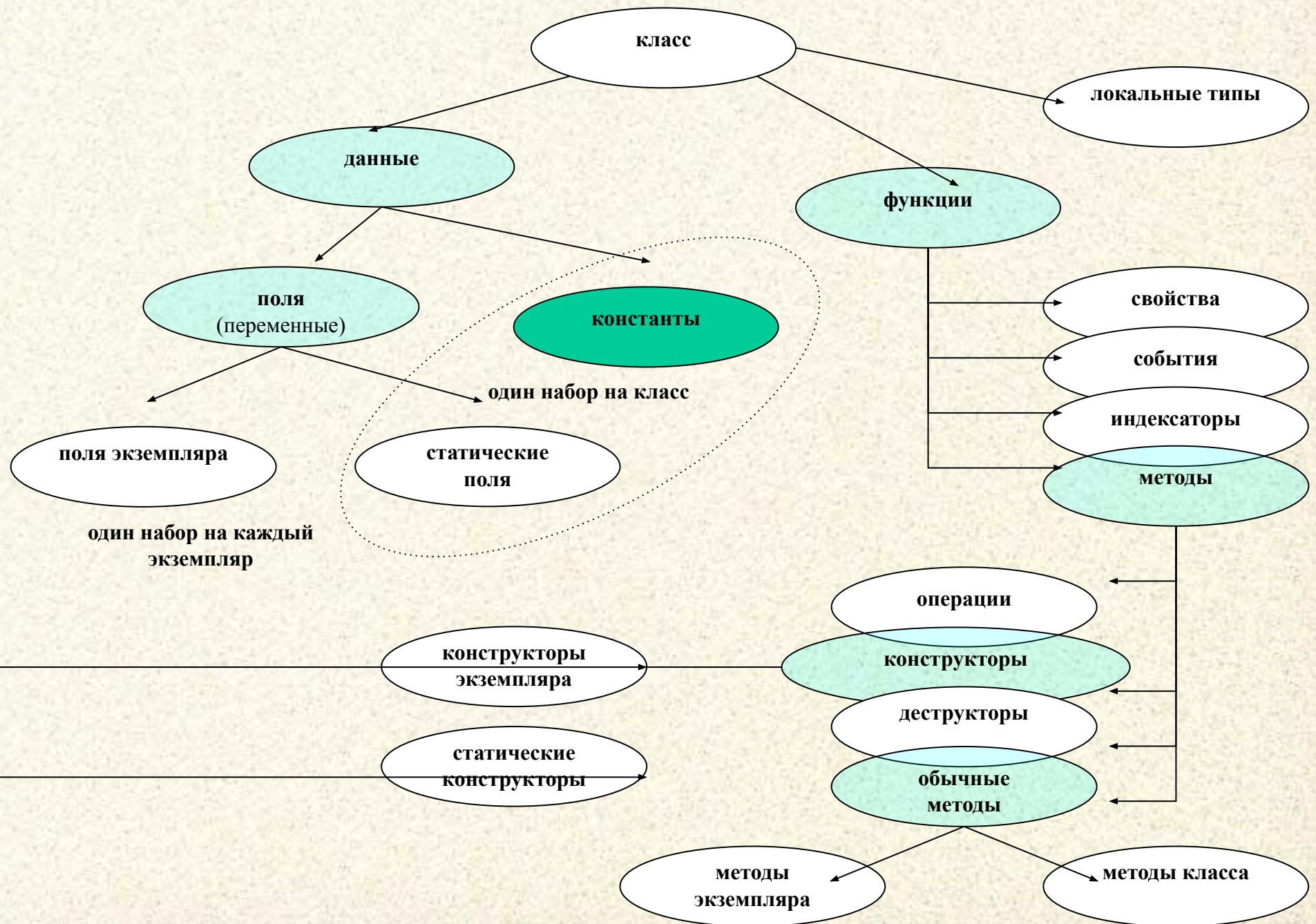
Простейший пример описания класса:

```
class Demo {           // пустой класс
```


Спецификаторы класса

Спецификатор	Описание
new	(для вложенных классов). Задаёт 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
<u>internal</u>	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 описания класса



Сквозной пример класса

```
class Monster {
    public Monster() // конструкторы
    {
        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 string GetName() // метод
    { return name; }
    public int GetAmmo() // метод
    { return ammo;}
}
```

```
public int Health { // свойство
    get { return health; }
    set { if (value > 0) health = value;
        else health = 0;
    }
}

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

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

string name; // поле
int health, ammo; // поле
}
```


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.

Пример создания объектов (экземпляров)

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

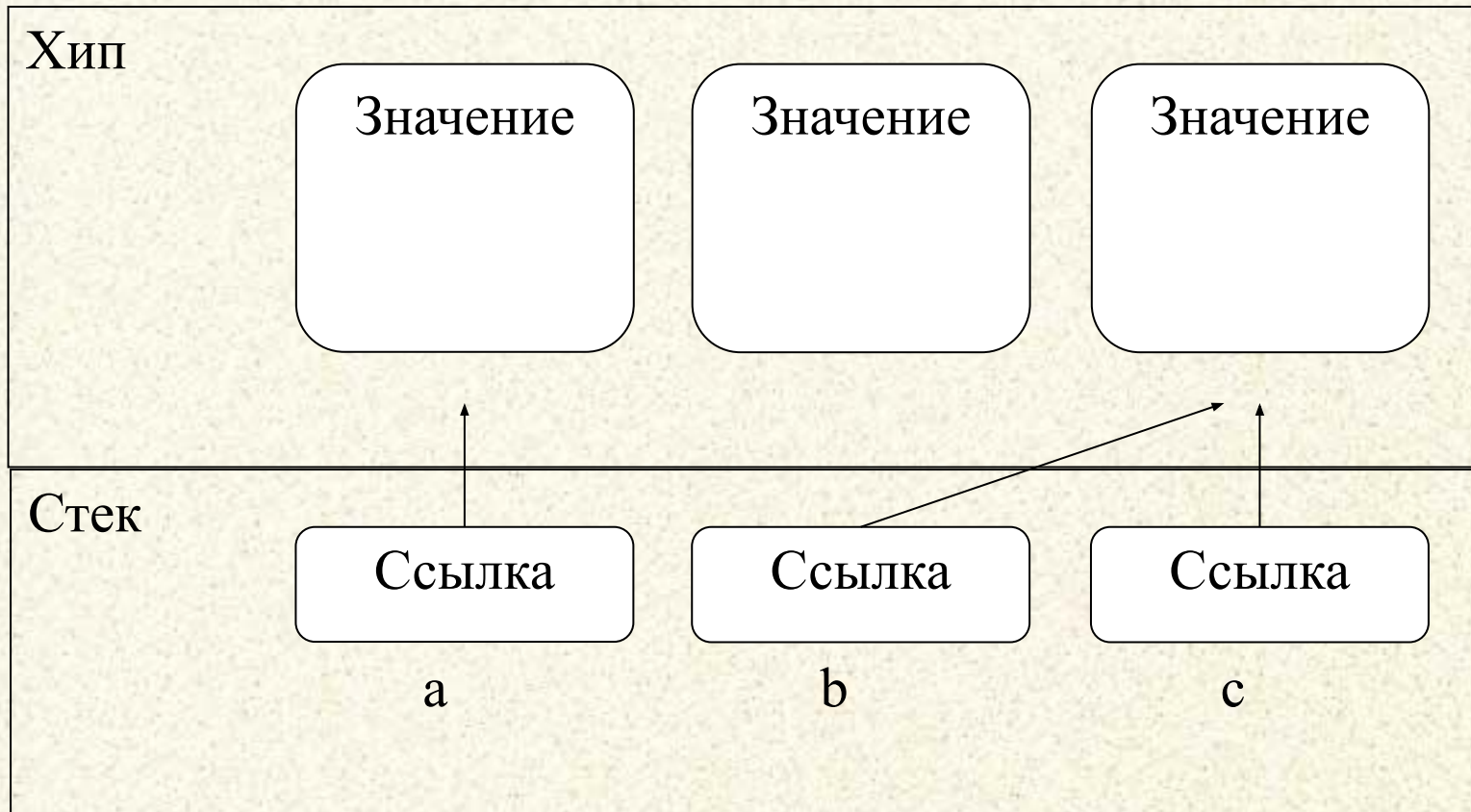
Результат работы программы:

Monster Noname health = 100 ammo = 100

Monster Вася health = 100 ammo = 100

Monster Маша health = 200 ammo = 200

Присваивание и сравнение объектов



Объекты

- $b = c$
- Величины ссылочного типа равны, если они ссылаются на одни и те же данные ($b == c$, но $a != b$ даже при равенстве их значений или если обе ссылки равны null).

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.

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
        }
    }
}
```


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

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


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 Sety( double z ) { // открытый метод класса  
        y = z;  
    }  
}  
  
... Demo x = new Demo();    // где-то в методе другого класса  
x.Sety(3.12); ...          // ВЫЗОВ МЕТОДА
```


Примеры методов

```
public void Sety(double z) {  
    y = z;  
}  
public double Gety() {  
    return y;  
}
```

Вызывающая
функция

Вызов метода

Метод

return [...]

- 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.*

Возврат
значения

```
x.Sety(3.12);  
double t = x.Gety();
```


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.

Пример

```
class Demo {
    public int a = 1;
    public const double c = 1.66;
    static string s = "Demo";
    double y;
    public double Gety() { return y; } // метод получения y
    public void Sety( double y_ ) { y = y_; } // метод установки y
    public static string Gets() { return s; } // метод получения s
}
class Class1 {
    static void Main()
    { Demo x = new Demo();
      x.Sety(0.12); // вызов метода установки y
      Console.WriteLine(x.Gety()); // вызов метода получения y
      Console.WriteLine(Demo.Gets()); // вызов метода получения s
// Console.WriteLine(Gets()); // вариант вызова из того же
}} // класса
```


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.

Описание объекта: `SomeObj obj = new SomeObj();`

Описание аргументов: `int b; double a, c;`

Вызов метода: `obj.P(a, b, c);`

Заголовок метода P: `public void P(double x, int y, double z);`

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
    }
}
```

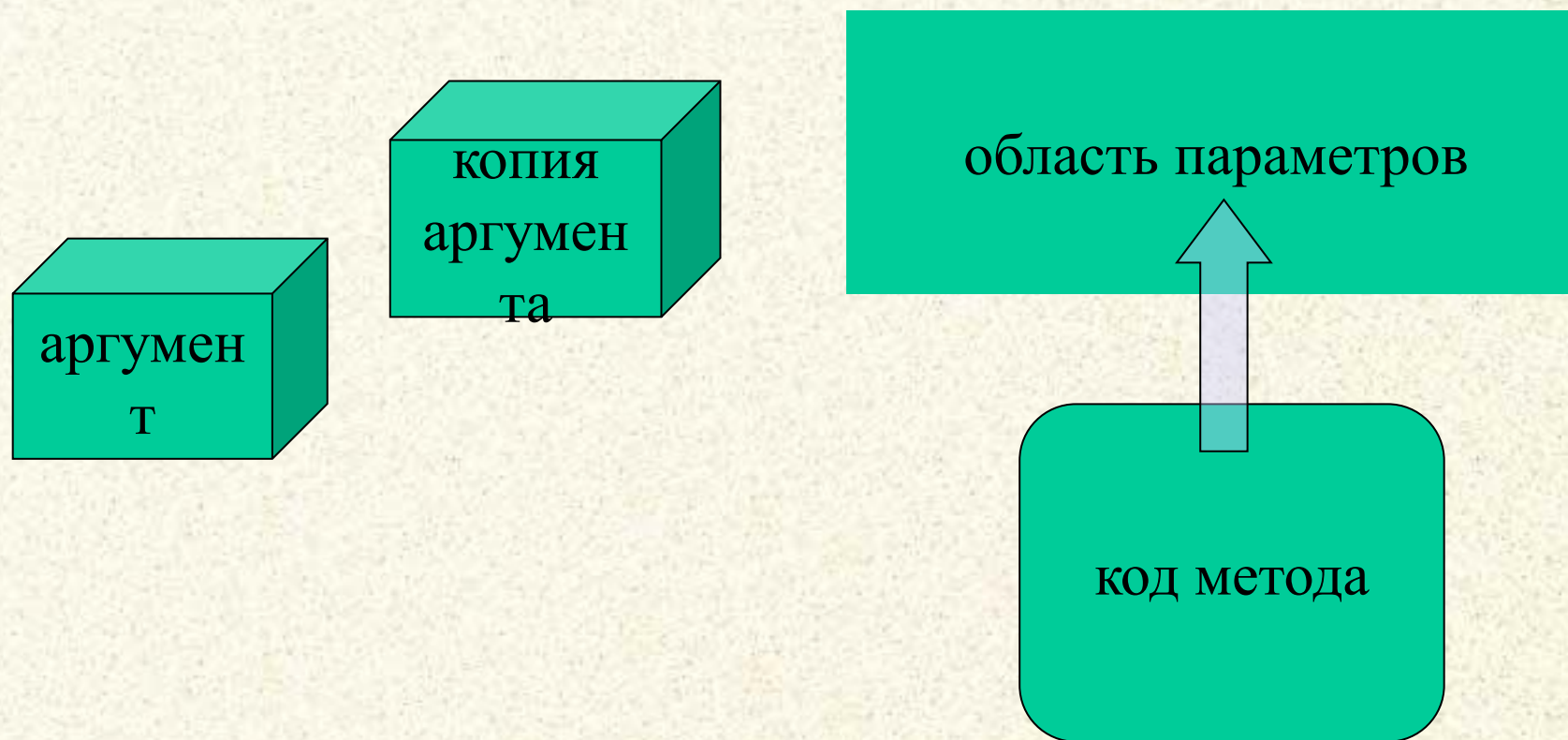

Способы передачи аргументов в метод

Аргументы передаются:

- По значению
- По адресу (ссылке)

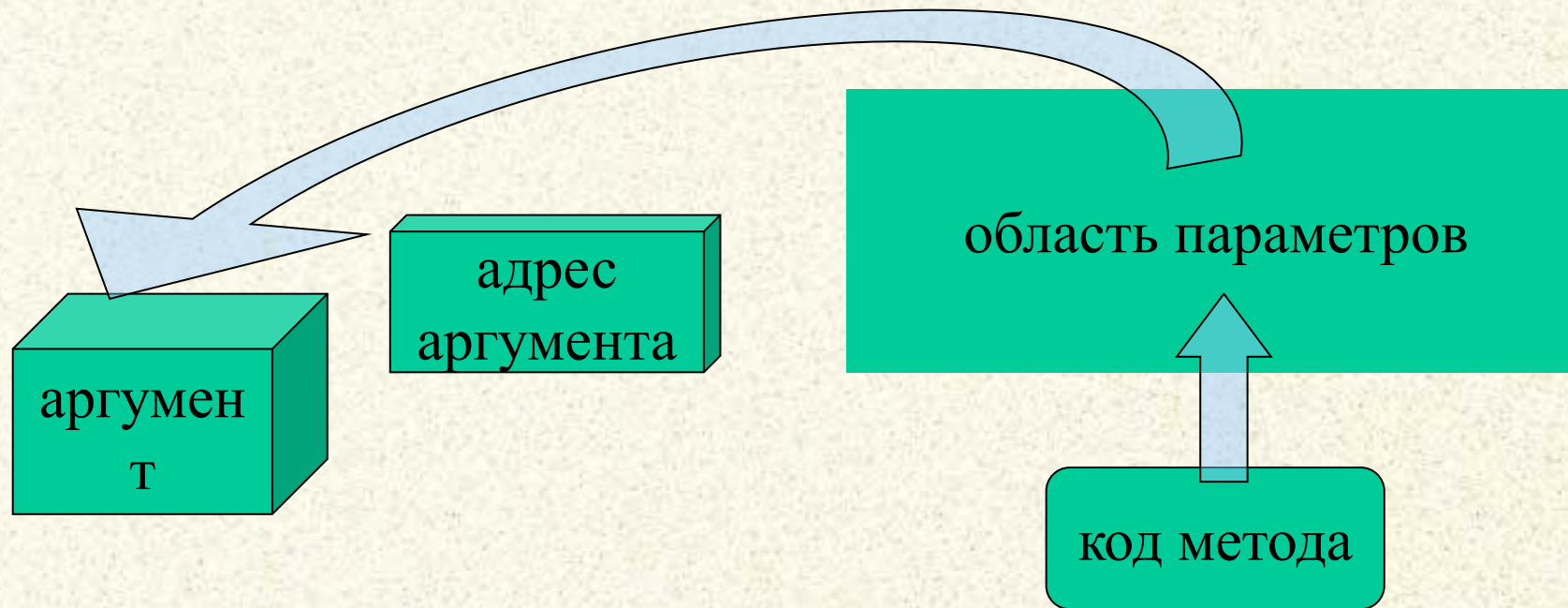
- *При передаче по значению* метод получает копии значений аргументов, и операторы метода работают с этими копиями.
- *При передаче по ссылке (по адресу)* метод получает копии адресов аргументов и осуществляет доступ к аргументам по этим адресам.

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.

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`.

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.

Пример:

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

параметр-
значение

параметр-
ссылка

выходной
параметр

параметр-
массив

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

1. Для параметров-значений используется передача по значению. Этот способ применяется для исходных данных метода.
 - 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.
2. Параметры-ссылки и выходные параметры передаются по адресу. Этот способ применяется для передачи побочных результатов метода.
 - 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**.

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 {
        int[] a = { 10, 20, 30 };
        Console.WriteLine( Average( a ) );           // 1
        int[] b = { -11, -4, 12, 14, 32, -1, 28 };
        Console.WriteLine( Average( b ) );           // 2
        short z = 1, e = 13;
        byte v = 100;
        Console.WriteLine( Average( z, e, v ) );     // 3
        Console.WriteLine( Average() );              // 4
    }
    catch( Exception e ) {Console.WriteLine( e.Message ); return; }
}}
```

Результат:

20

10

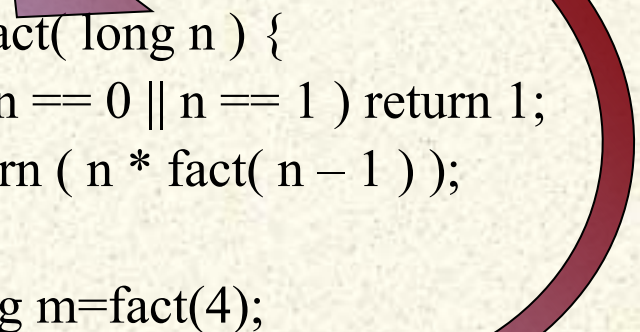
38

Недостаточно аргументов

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; // не рекурсивная 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; }
```

СТЕК

n = 1 ...

n = 2 ...

n = 3 ...

n = 4 ...

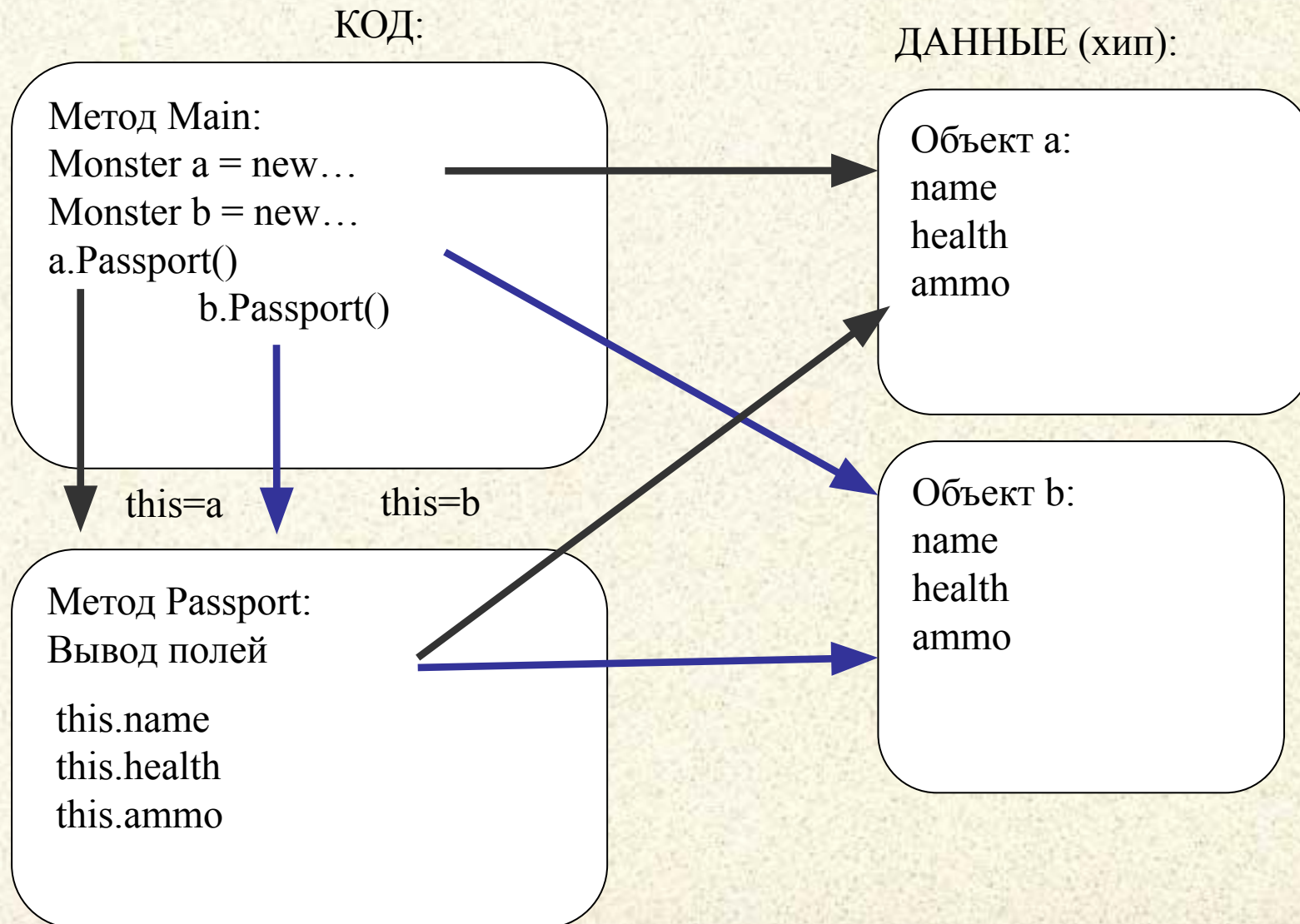
Характеристики рекурсии

Достоинство рекурсии: компактность записи.

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

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.



Использование явного this

В явном виде параметр `this` применяется:

1) чтобы вернуть 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 Sety( double y ) { this.y = y; }
}
```


Конструкторы

Конструктор предназначен для инициализации объекта. Он 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
```


Сквозной пример класса

```
class Monster {  
    public Monster() // конструктор  
    {  
        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 string GetName() // метод  
    { return name; }  
    public int GetAmmo() // метод  
    { return ammo;}
```

```
    public int Health { // СВОЙСТВО  
        get { return health; }  
        set { if (value > 0) health = value;  
            else health = 0;  
        }  
    }  
  
    public void Passport() // метод  
    { Console.WriteLine(  
        "Monster {0} \t health = {1} \  
        ammo = {2}", name, health, ammo  
    );  
    }  
  
    public override string ToString(){  
        string buf = string.Format(  
            "Monster {0} \t health = {1} \  
            ammo = {2}", name, health,  
            ammo);  
        return buf; }  
  
    string name;  
    int health, ammo;  
}
```


*Любой человек должен уметь менять пеленки,
планировать вторжения, резать свиней,
конструировать здания, управлять кораблями,
писать сонеты, вести бухгалтерию, возводить
стены, вправлять кости, облегчать смерть,
исполнять приказы, отдавать приказы,
сотрудничать, действовать самостоятельно,
решать уравнения, анализировать новые
проблемы, бросать навоз, программировать
компьютеры, вкусно готовить, хорошо
сражаться, достойно умирать.
Специализация — удел насекомых.*

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 описания свойств

```
public class Button: Control
{ private string caption;           // поле, с которым связано свойство
  public string Caption {           // свойство
    get { return caption; }        // способ получения свойства

    set                             // способ установки свойства
      { if (caption != value) { caption = value; }
  }
  ...
}
```

В программе свойство выглядит как поле класса:

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


Сквозной пример класса

```
class Monster {  
    public Monster() // конструктор  
    {  
        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 string GetName() // метод  
    { return name; }  
    public int GetAmmo() // метод  
    { return ammo; }  
}
```

```
public int Health { // свойство  
    get { return health; }  
    set { if (value > 0) health = value;  
        else health = 0;  
    }  
}  
public string Name { // свойство  
    get { return name; }  
}  
public void Passport() // метод  
    { Console.WriteLine(  
        "Monster {0} \t health = {1} \  
        ammo = {2}", name, health, ammo );  
    }  
public override string ToString(){  
    string buf = string.Format(  
        "Monster {0} \t health = {1} \  
        ammo = {2}", name, health, ammo);  
    return buf; }  
  
string name;  
int health, ammo;  
}
```


Пример описания свойств

```
namespace ConsoleApplication1
```

```
{ class Ball {  
    public Ball(double radius, double density)  
    { this.radius = radius;  
      this.density = density;  
    }  
  
    public double Mass  
    { get { return radius * density; } }  
  
    public double Diameter  
    { get { return radius * 2; } }  
  
    double radius;  
    double density;  
}
```

```
class Program  
{ static void Main(string[] args)  
  {  
    Ball ball = new Ball(10, 6.2);  
    Console.WriteLine("Диаметр: " +  
                      ball.Diameter +  
                      " Масса: " + ball.Mass);  
  }  
}
```


Перегрузка методов

- Использование нескольких методов с одним и тем же именем, но различными типами параметров называется *перегрузкой методов*.
- 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*

Сквозной пример класса

```
class Monster {  
    public Monster() // конструктор  
    {  
        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 string GetName() // метод  
    { return name; }  
    public int GetAmmo() // метод  
    { return ammo;}
```

```
public int Health { // СВОЙСТВО  
    get { return health; }  
    set { if (value > 0) health = value;  
        else health = 0;  
    }  
}  
  
public void Passport() // метод  
    { Console.WriteLine(  
        "Monster {0} \t health = {1} \  
        ammo = {2}", name, health, ammo );  
    }  
  
public override string ToString(){  
    string buf = string.Format(  
        "Monster {0} \t health = {1} \  
        ammo = {2}", name, health, ammo);  
    return buf; }  
  
string name;  
int health, ammo;  
}
```


Операции класса

- 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.

Общие правила описания операций класса

- 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).

Унарные операции

Можно определять: + - ! ~ ++ -- true false

Примеры заголовков:

```
public static int operator +( MyObject m ) // унарный плюс
```

```
public static MyObject operator --( MyObject m ) // декремент
```

```
class Monster {  
    public static Monster operator ++(Monster m)  
    {  
        Monster temp = new Monster();  
        temp.health = m.health + 1;  
        return temp;  
    }  
    ...  
}  
...  
Monster vasia = new Monster();  
++vasia; vasia++;  
...
```


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


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
```


Применение операций преобразования

- *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.

Summary

Интерфейс класса

- При создании класса следует хорошо продумать его *интерфейс* — средства работы с классом, доступные использующим его программистам.
- Интерфейс хорошо спроектированного класса интуитивно ясен, непротиворечив и обозрим. Как правило, он не должен включать поля данных.
- В идеале *интерфейс должен быть полным* (предоставлять возможность выполнять любые разумные действия с классом) и *минимально необходимым* (без дублирования и пересечения возможностей методов).

Состав класса

- Как правило, класс как тип, определенный пользователем, должен содержать скрытые (*private*) поля и следующие функциональные элементы:
 - *конструкторы*, определяющие, как инициализируются объекты класса;
 - набор *методов* и *свойств*, реализующих характеристики класса;
 - классы *исключений*, используемые для сообщений об ошибках путем генерации исключительных ситуаций.
 - Классы, моделирующие математические или физические понятия, обычно также содержат набор *операций*, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия, требующиеся по сути класса.

Элементы класса

- **Методы** определяют поведение класса. *Каждый метод класса должен решать только одну задачу.*
- *Создание любого метода следует начинать с его интерфейса (заголовка).* Необходимо четко представлять себе, какие параметры метод должен получать и какие результаты формировать. Входные параметры обычно перечисляют в начале списка параметров.
- **Поля**, характеризующие класс в целом, следует описывать как *статические*.
- Все литералы, связанные с классом, описываются как поля-константы с именами, отражающими их смысл.
- Необходимо стремиться к максимальному сокращению области действия каждой переменной. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки.

Далее...

- Событийное программирование
- Работа с памятью
- Визуальное проектирование и элементы управления
- Работа с файлами
- Работа с реляционными СУБД