Общее понятие ООП. Принципы ООП: наследование, инкапсуляция, полиморфизм.

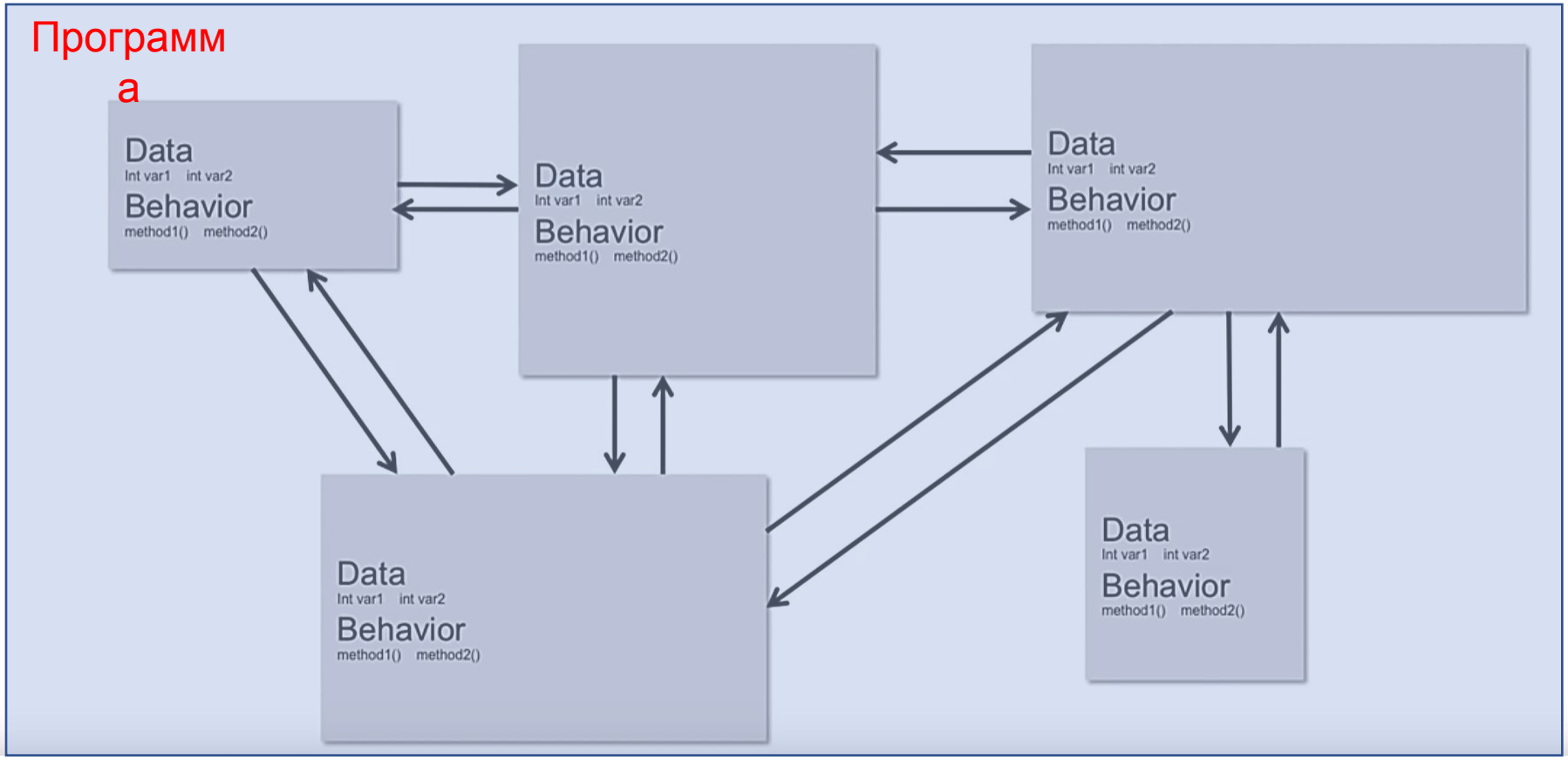
Классы: основные понятия

Основные элементы класса: поля, методы, конструкторы, свойства.

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.

Общее понятие ООП



При создании объектно-ориентированной программы предметная область представляется в виде совокупности объектов.

Выполнение программы состоит в том, что объекты обмениваются сообщениями.

Определение

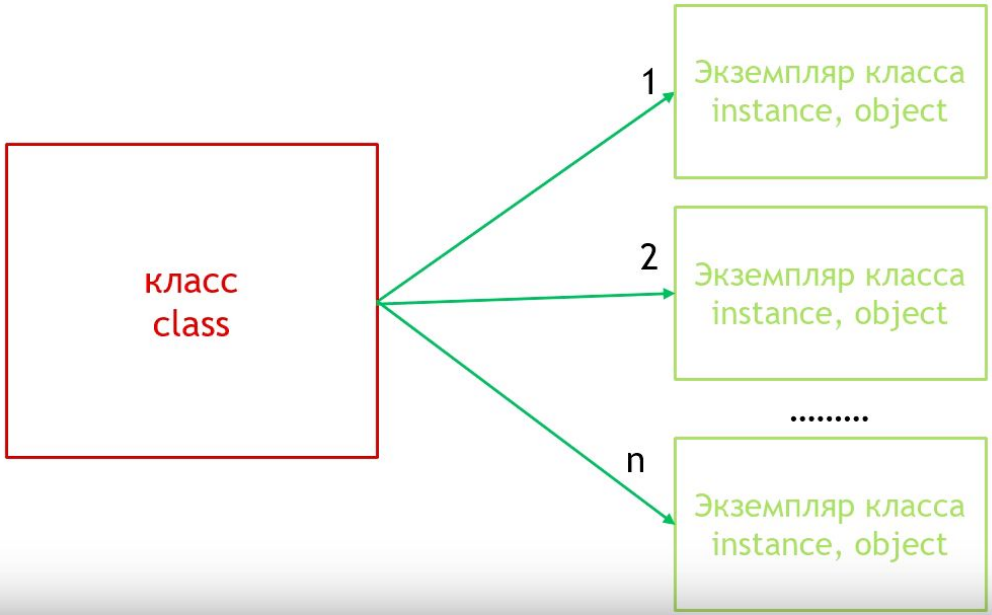
- **Объектно-ориентированное программирование** (ООП, Object-Oriented Programming) - совокупность принципов, технологий , а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия **объектов**.

Основные определения ООП

Объект в ООП является экземпляром того или иного класса.

- **Класс** представляет собой множество **объектов**
  - имеющих общую структуру
  - обладающих одинаковым поведением.

- Экземпляр (объект) класса – каждый такой объект, конкретная сущность.



Классы объектов

Классом называют особую структуру, которая может иметь в своем составе поля, методы и свойства.

Класс
Cats

Члены класса (class members)

ПОЛЯ

```
//ПОЛЯ (FIELDS)
string name;
int age;
string color;
```

МЕТОДЫ

```
//МЕТОДЫ (METHODS)
0 references
public string Mau()
{
    return name + " Мяуууу!";
}
```

СВОЙСТВА

```
//СВОЙСТВА (PROPERTIES)
0 references
public string Name
{
    get;
    set;
}
```

Мурзик
1 год
Рыжий

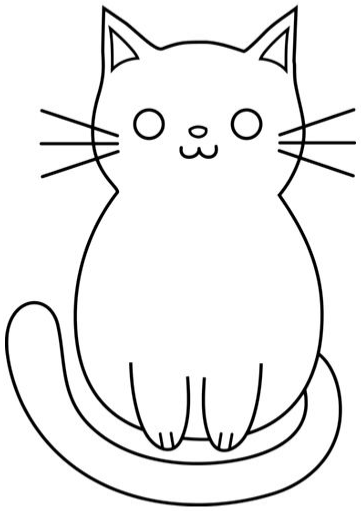
КОНСТРУКТОРЫ

```
public Cats()    //КОНСТРУКТОР по умолчанию
{

}

//КОНСТРУКТОР с параметрами
0 references
public Cats(string name1, int age1, string color1)
{
    name = name1;
    age = age1;
    color = color1;
}
```

Классы и объекты

Пример 1

CLASS Cat



Имя
Цвет
Возраст

Мяукать

objects

1

Мурзик
Рыжий
1 год

2

Борис
Черный
8 лет

3

Пушок
Белый
2 года

Классы и объекты

Пример 2

CLASS Card

Ранг
Масть

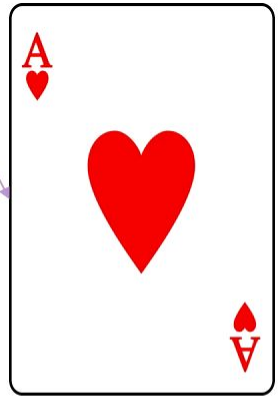
Сбросить

objects

1

Валет
Пики

2

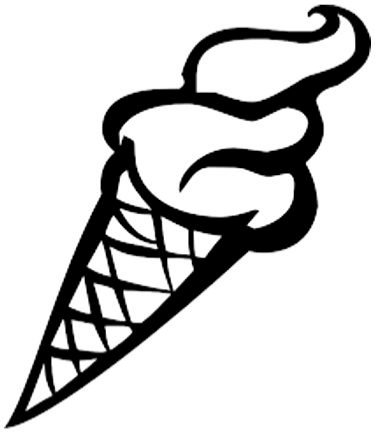


Туз
Черви

Классы и объекты

Пример 3

CLASS IceCreamInCone



Вкус
Количество шариков

Купить
Съесть

objects

Клубничный
1

Шоколадный
1

Шоколадный
2

Мятный
2

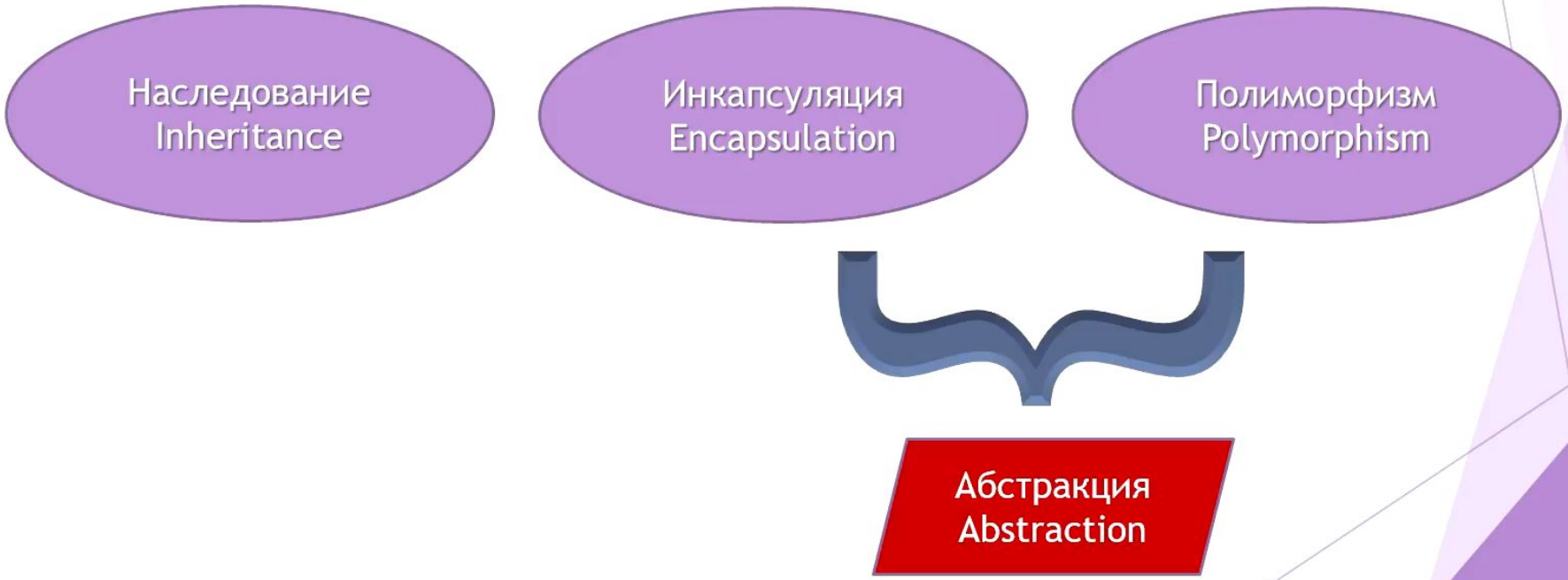
Класс и объект

Класс и объект

```
class MyClass {

   String name = "Example";

     // "Конструктор"
     public MyClass(String name) {
         this.name = name;
   }
     // "Метод"
     public String getName() {
         return name;
   }
}
```

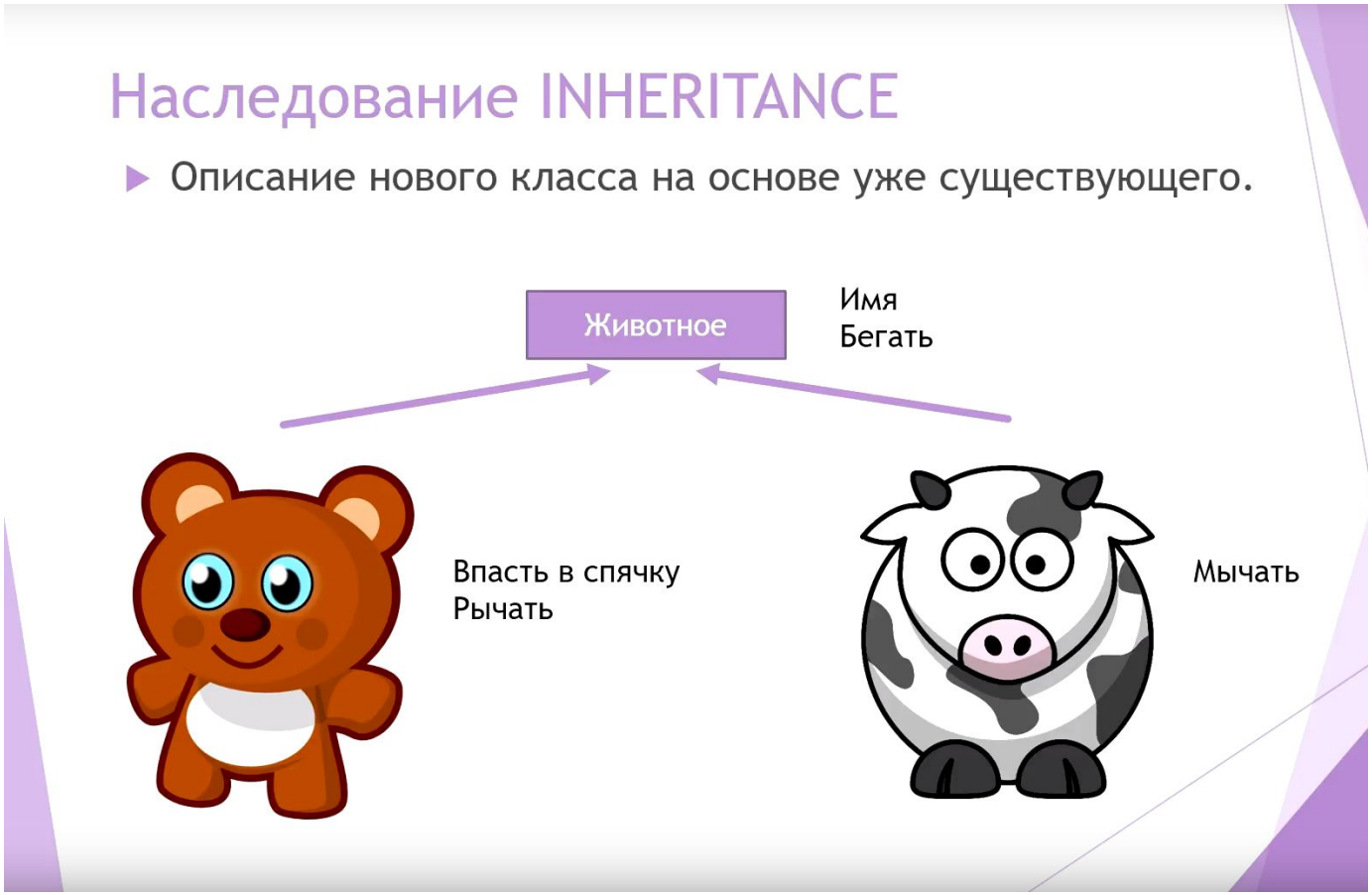
Основные принципы ООП

ООП
Object-oriented programming



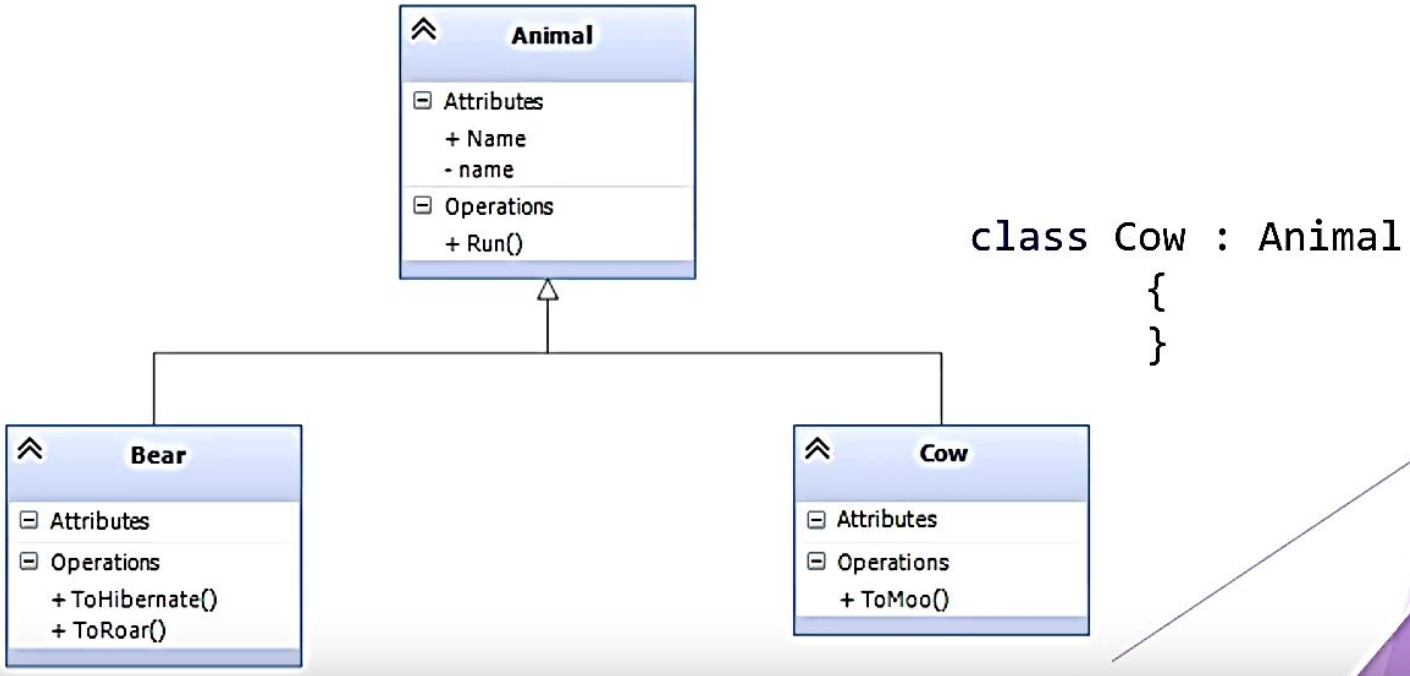
Наследование

Наследование (inheritance) — это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование), или других (множественное наследование) классов.



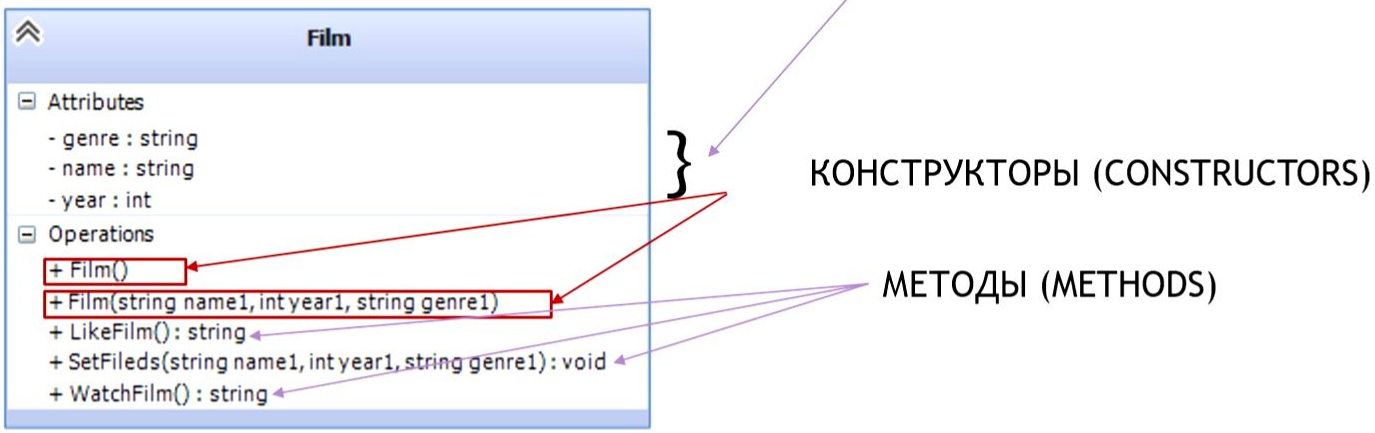
Наследование INHERITANCE

- Базовый класс = родительский класс (base class / parent class / superclass);
- Дочерний класс = класс-наследник (subclass / inherited class);
- Дочерний класс наследует (derived from / inherits / extends) базовый класс.




```
class Cow : Animal
        {
        }
```

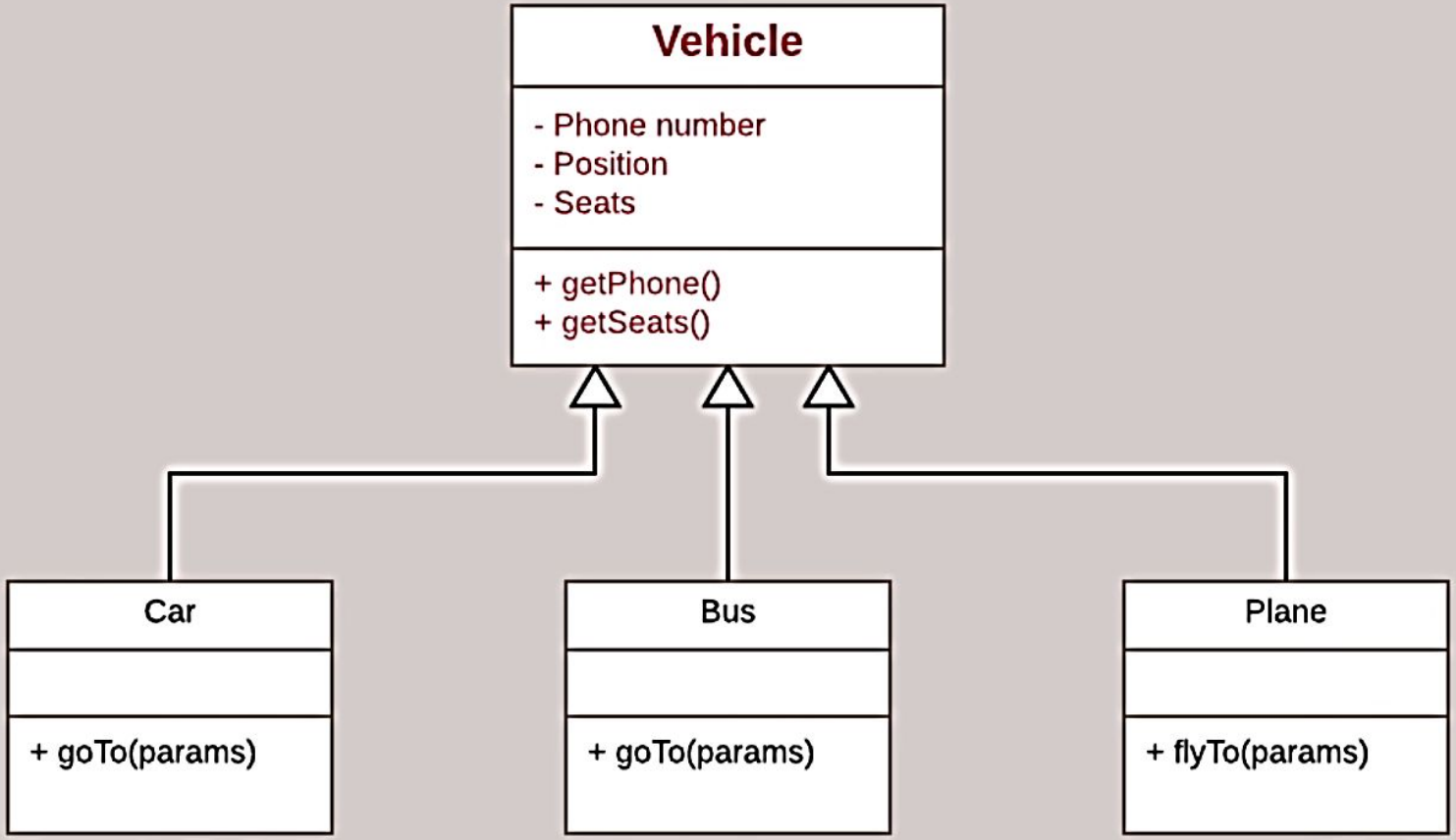
UML-диаграмма класса



+ public
- private

Термины обозначающие механизм наследования

Базовый	Наследник
Родительский	Дочерний
Предок	Потомок
Надкласс	Подкласс
Супер класс	Суб класс


Vehicle
- Phone number
- Position
- Seats
+ getPhone()
+ getSeats()
Car
+ goTo(params)
Bus
+ goTo(params)
Plane
+ flyTo(params)

Наследование

Наследование применяется для:

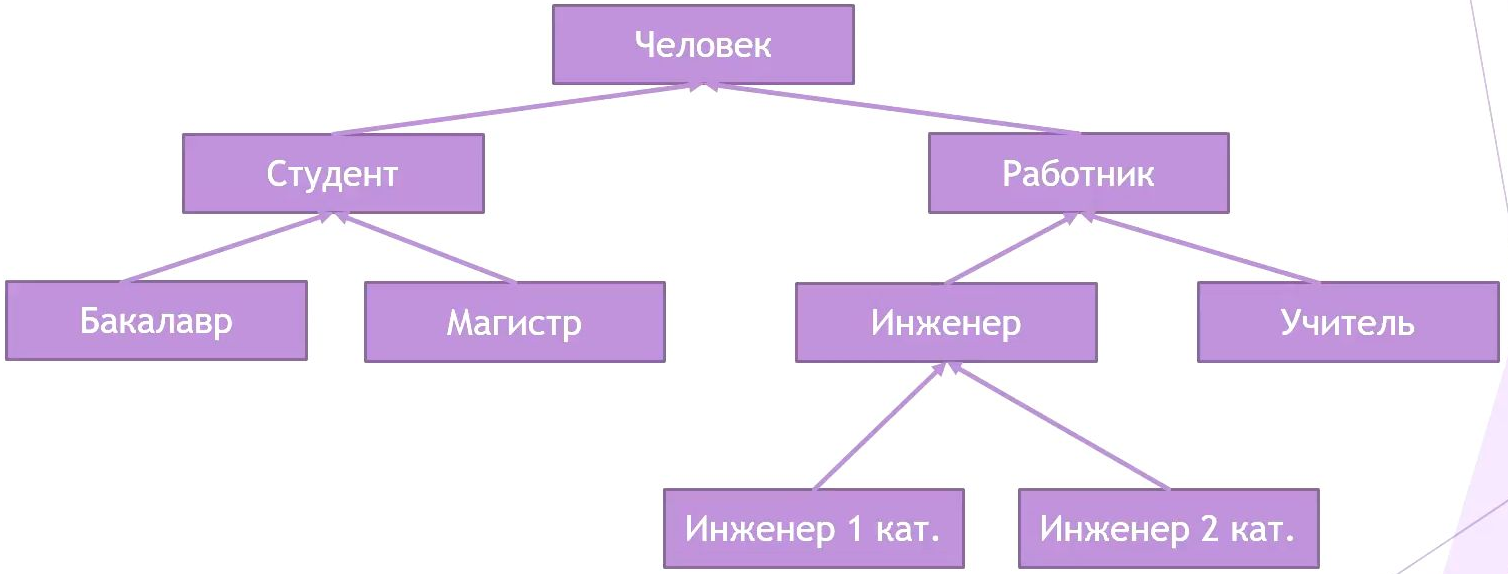
- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

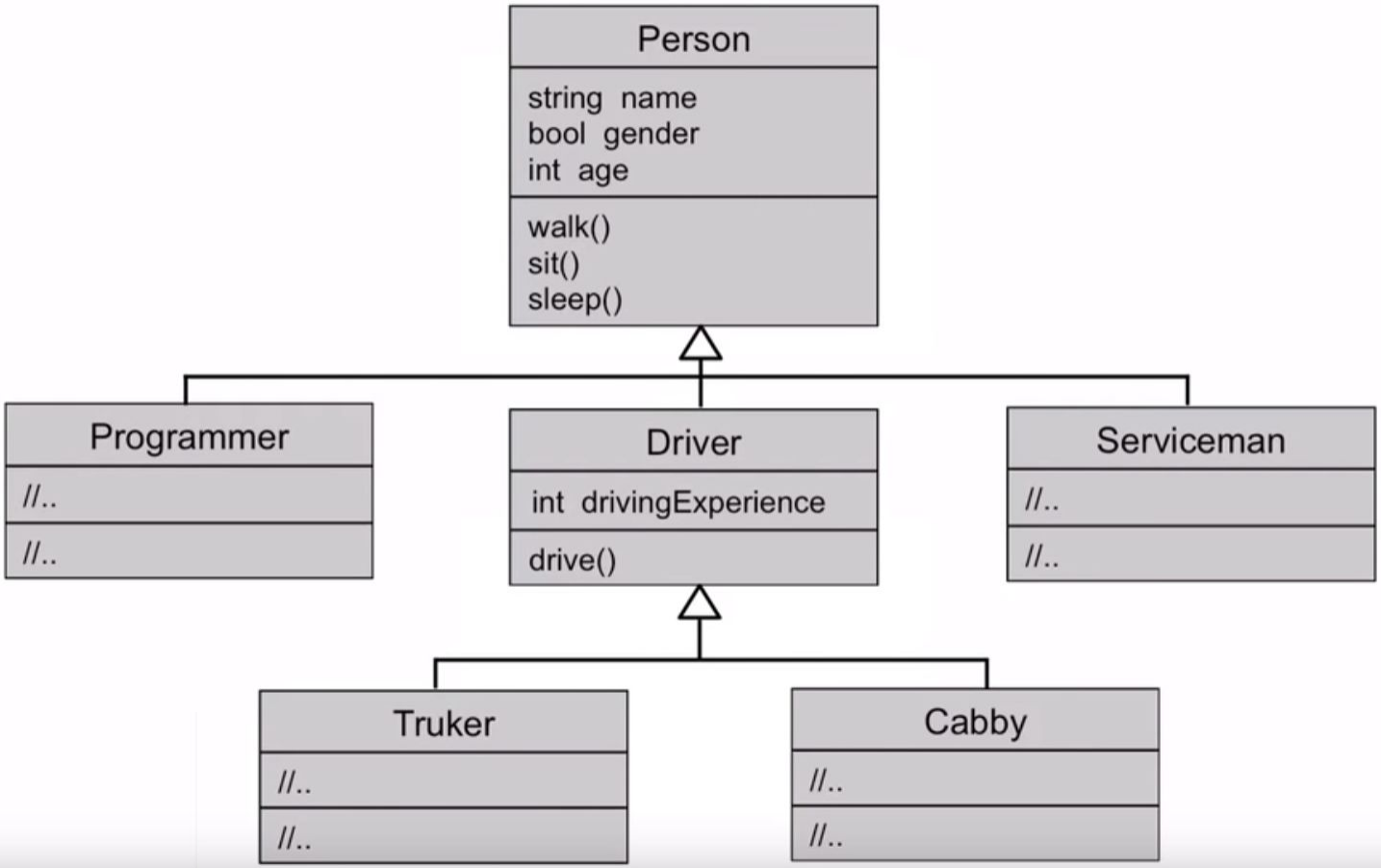
Наследование позволяет создавать *иерархии классов*. Иерархия представляется в виде дерева, в котором более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях.

Иерархия классов

Наследование INHERITANCE



Иерархия классов



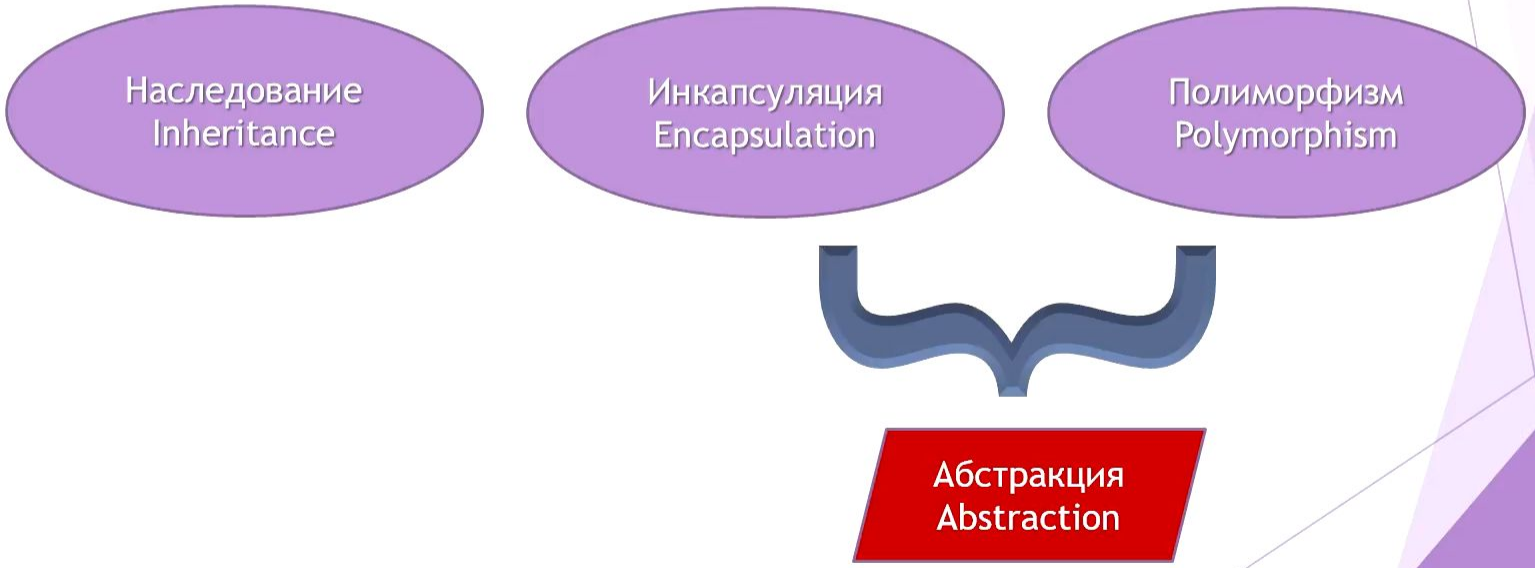
ЗАПОМНИТЬ

- Наследование - описание нового класса на основе уже существующего.
- Класс C# может иметь только **1** класс-родитель, но реализовывать ∞ (множество) интерфейсов.
- У класса-родителя может быть сколько угодно наследников.
- Всё, что может делать базовый класс, также может делать и дочерний.

```
class Cat : Animal
      {
      }
```

Основные принципы ООП

ООП
Object-oriented programming



Что такое инкапсуляция?

Что такое инкапсуляци

публичные

интерфейс класса

инкапсулированные

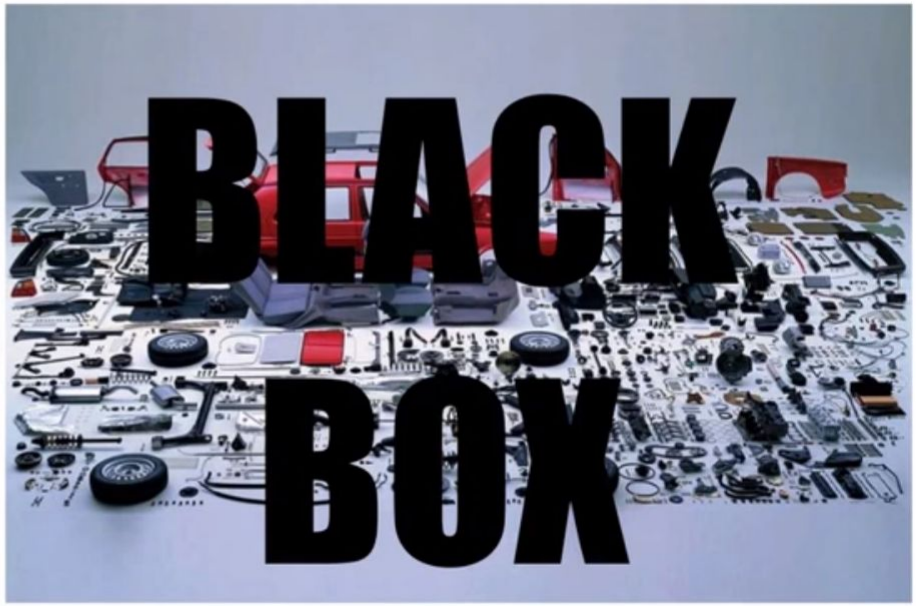
скрытая реализация класса

Что такое инкапсуляция?

публичные

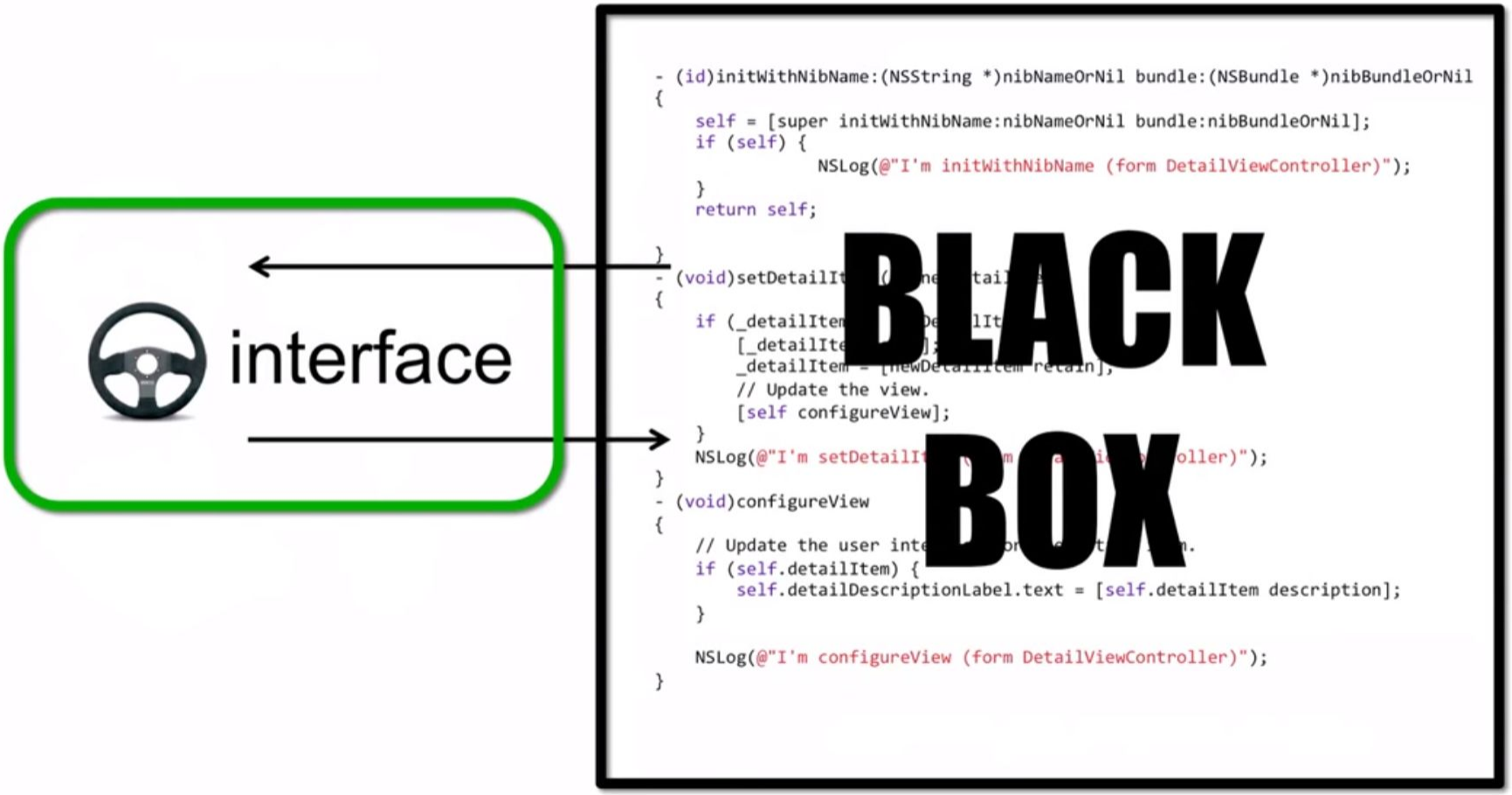
интерфейс класса

инкапсулированные



скрытая реализация класса

Что такое инкапсуляция?

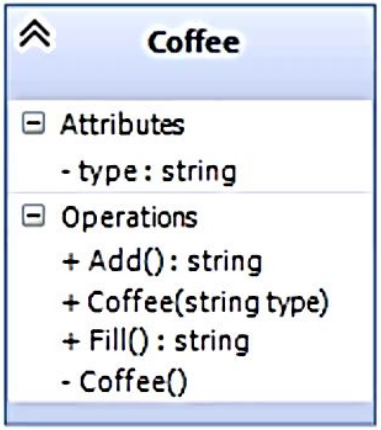


Инкапсуляция

- 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.

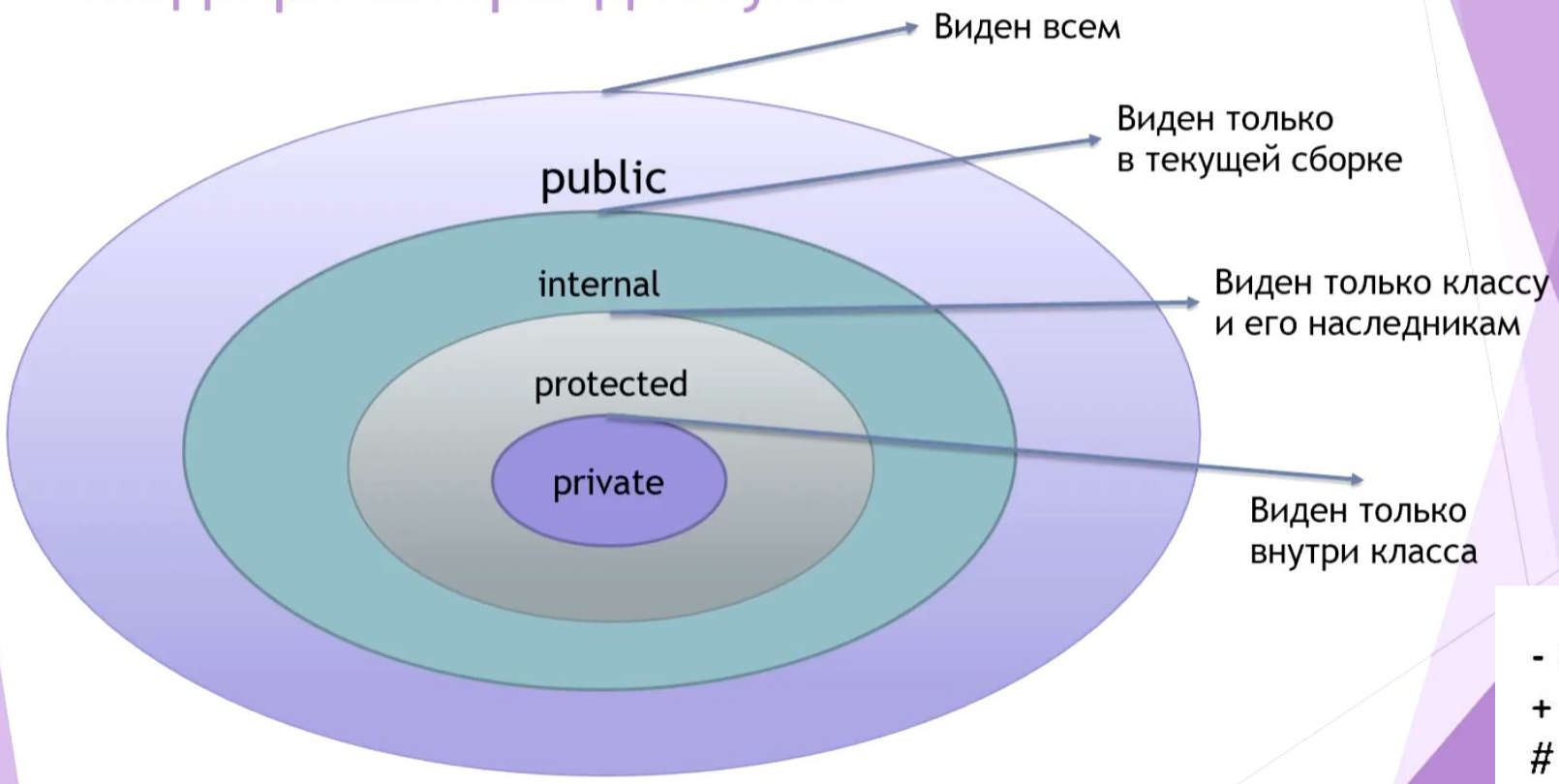
Инкапсуляция Encapsulation

- Сокрытие членов типа внутри типа



- private
+ public
# protected
~ internal

Модификаторы доступа



- private
+ public
# protected
~ internal

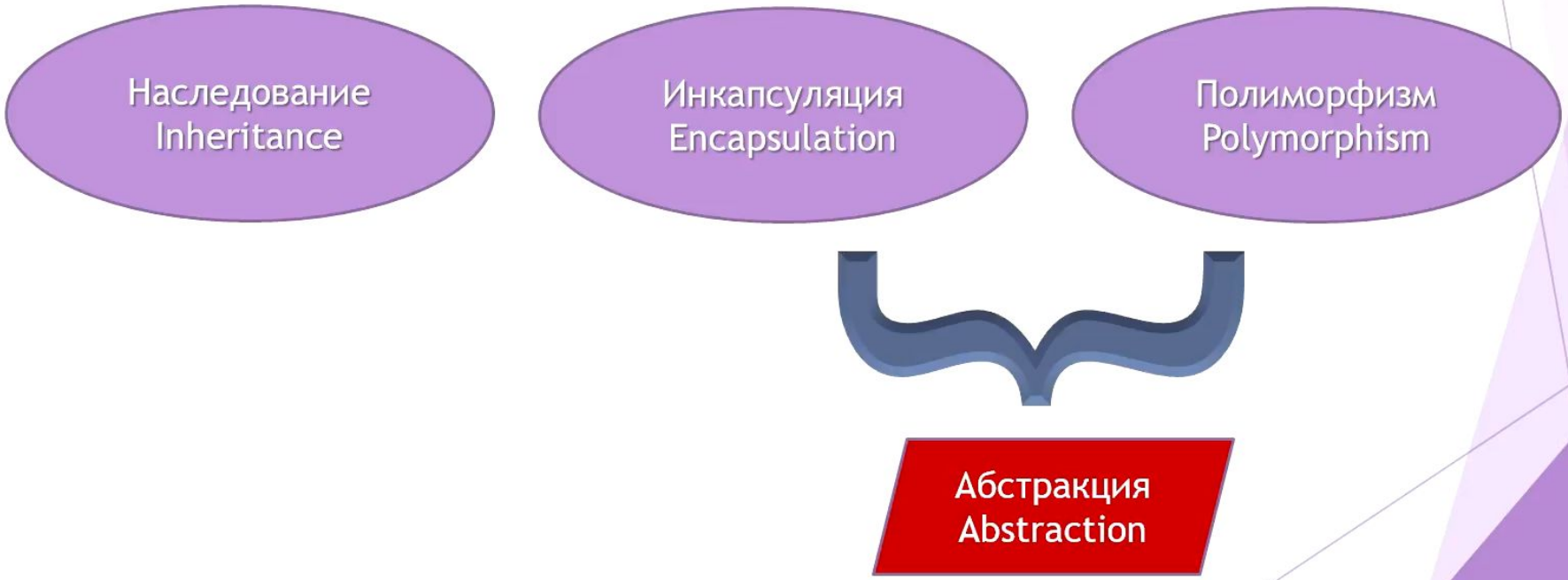
ЗАПОМНИТЬ

- Инкапсуляция – сокрытие членов типа внутри типа.
- Инкапсуляция реализуется с помощью модификаторов доступа: public, private, protected, internal.
- Все члены класса по умолчанию – private.
- Классы по умолчанию имеют модификатор internal.


Класс
Данные
Код

Основные принципы ООП

ООП
Object-oriented programming



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**.
- Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Полиморфизм Polymorphism

Статический

Перегрузка
Overloading

Динамический

Переопределение
Overriding

Перегрузка (overloading)

- Использование в одном классе методов с одинаковым названием, но разными входными параметрами и типом возвращаемого значения.
  - Перегрузка конструкторов
  - Свойства перегружать нельзя

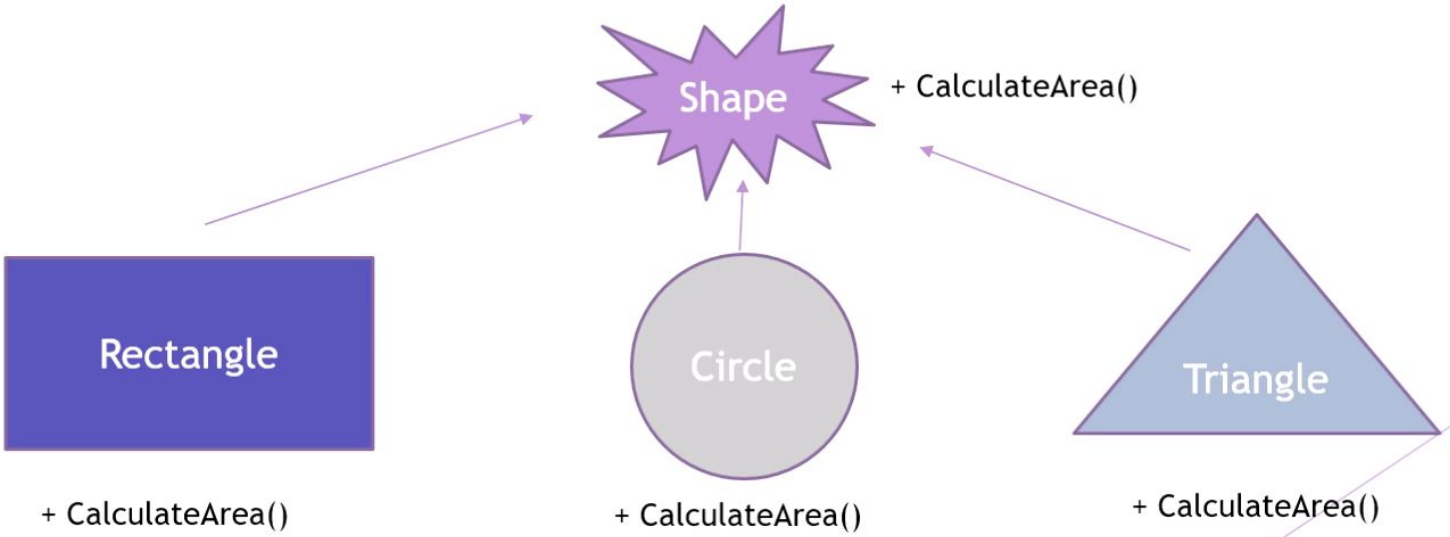
```
public void DrinkCoffee()
  {
     Console.WriteLine("Выпить кофе");
  }
```

```
public string DrinkCoffee(int sugarCount)
  {
     return "Кофе с " + sugarCount + " кусочками сахара";
  }
```

```
public bool DrinkCoffee(int sugarCount, double milk)
  {
     return true;
  }
```

Переопределение (overriding)

- Использование в классах-наследниках виртуальных/абстрактных методов родительского класса с одинаковым названием, входными параметрами и типом возвращаемого значения, но разной реализацией.
- Переопределенный метод в классе-наследнике всегда обозначается модификаторами `override` или `new`



Полиморфизм

Перегрузка vs Переопределение

- 1 класс
- Методы, конструкторы
- Одинаковое имя
- Разные параметры
- Разный тип возвращаемого значения (необязательно)
- Любой модификатор доступа

- Иерархия наследования
- Методы, свойства
- Одинаковое имя
- Одинаковые параметры
- Одинаковый тип возвращаемого значения
- Модификатор доступа должен совпадать

Пример непереопределяемого метода

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public virtual void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
class Employee : Person
{
    public string Company
    public Employee(string
        : base(firstName,
    {
    }
}
```

```
Bill Gates
Tom Smith
```

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill", "Gates");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Smith", "Microsoft");
    p2.Display(); // вызов метода Display из класса Person

    Console.ReadKey();
}
```

Пример переопределяемого метода

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public virtual void Display()
    {
        Console.WriteLine($"{First
    }
}
```

```
class Employee : Per
{
    public string Co
    public Employee(
        : base(first
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill", "Gates");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Smith", "Microsoft");
    p2.Display(); // вызов метода Display из класса Employee

    Console.ReadKey();
}
```

```
Bill Gates
Tom Smith работает в Microsoft
```

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public virtual void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
```

Родительский класс

Класс - наследник

Вызов метода из класса - родителя

Вызов метода из класса - наследника

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill", "Gates");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Smith", "Microsoft");
    p2.Display(); // вызов метода Display из класса Employee

    Console.ReadKey();
}
```

```
Bill Gates
Tom Smith работает в Microsoft
```

ЗАПОМНИТЬ

- Полиморфизм = Перегрузка + Переопределение
- Перегрузка (overloading) - использование в одном классе методов с одинаковым названием, но разными входными параметрами и типом возвращаемого значения.
- Переопределение (overriding) - использование в классах-наследниках виртуальных/абстрактных методов базового класса с одинаковым названием, входными параметрами и типом возвращаемого значения, но разной реализацией.

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;
- неграмотное применение ООП может привести к значительному ухудшению характеристик разрабатываемой программы.

Технология разработки ОО программ

В процесс проектирования добавляется еще один этап - разработка иерархии классов.

1. в предметной области выделяются понятия, которые можно использовать как классы. Кроме классов из прикладной области, появляются классы, связанные с аппаратной частью и реализацией
2. определяются операции над классами, которые впоследствии станут методами класса. Их можно разбить на группы:

   - связанные с конструированием и копированием объектов

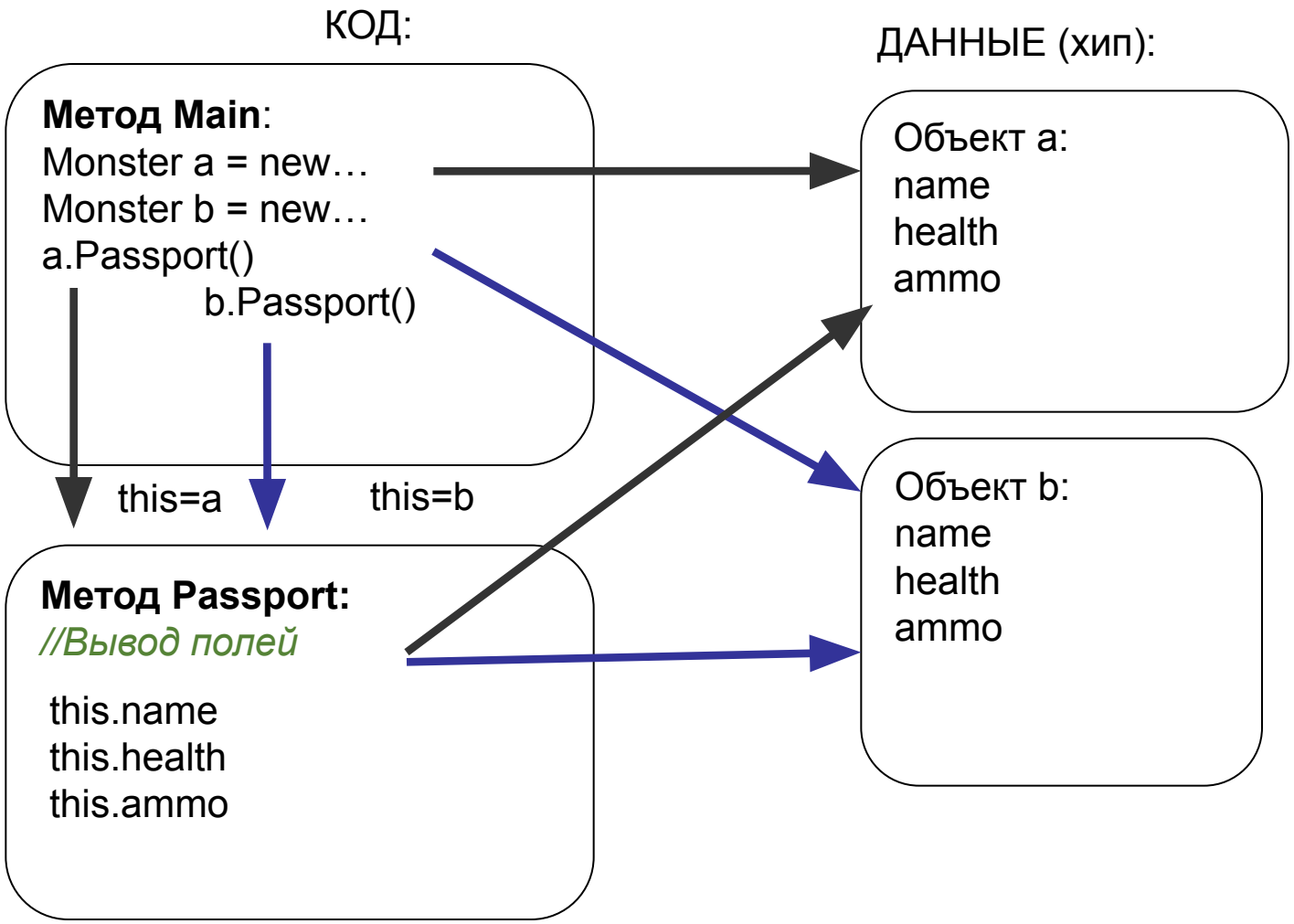
   - для поддержки связей между классами, которые существуют в прикладной области

   - 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.

Классы должны как можно ближе соответствовать моделируемым объектам из предметной области.

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.



Ключевое слово this

- Указатель **this** - это указатель на объект, для которого был вызван нестатический метод.
- Ключевое слово ***this*** обеспечивает доступ к текущему экземпляру класса.
- Классический пример использования *this*, это как раз в конструкторах, при одинаковых именах полей класса и аргументов конструктора.
- Ключевое слово *this* это что-то вроде имени объекта, через которое мы имеем доступ к текущему объекту.

```
namespace next_class
{
    public class Person
    {
        // Поля
        string firstName;
        string lastName;
        // Первый метод-конструктор
        public Person()
        {
            firstName = "Nike";
            lastName = "Black";
        }

        // Второй метод-конструктор
        public Person(string f, string l)
        {
            this.firstName = f;
            this.lastName = l;
        }
    }
}
```

Использование явного this

В явном виде параметр this применяется:

```
// чтобы возвратить из метода ссылку на вызвавший объект
class Demo
{      double y;
       public Demo T()    { return this; }
// для идентификации поля, если его имя совпадает с именем параметра
   метода
       public void Sety( double y ) { this.y = y; }
}
```

Обобщение понятия класса

Элементы класса

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

Простейший пример описания класса:

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

Спецификаторы класса

Спецификатор	Описание
new	Используется 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	Используется 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. Введен в версию языка 2.0.

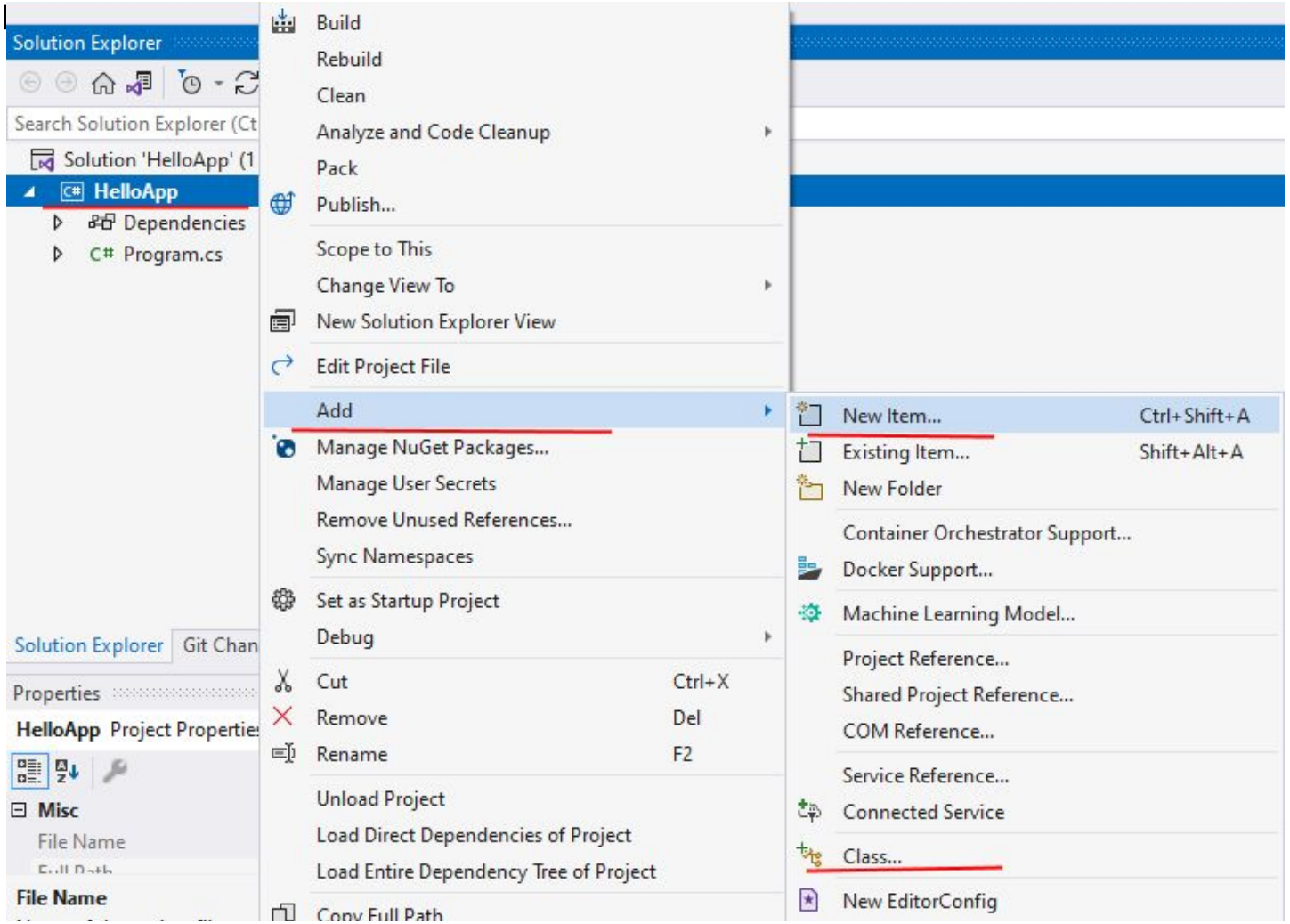
Сквозной пример класса

```
1.  class Monster {
2.    public Monster()
3.    {
4.          this.name  = "Noname";
5.          this.health = 100;
6.          this.ammo = 100;
7.    }
8.   public Monster( string name ) : this()
9.    {
10.         this.name = name;
11.    }
12.  public Monster( int health, int ammo, string
     name )
13.    {
14.         this.name  = name;
15.         this.health = health;
16.         this.ammo = ammo;
17.    }
18.  public int GetName()
19.    { return name;  }
20.  public int GetAmmo()
21.    { return ammo;}
```

```
22.    public int Health
23.    {
24.      get { return health;  }
25.      set { if (value > 0) health = value;
26.            else  health = 0;
27.          }
28.    }
29.    public void Passport()
30.       {  Console.WriteLine("Monster {0} \t health = {1} \ t
       ammo = {2}", name, health, ammo );
31.       }
32.    public override string ToString()
33.    {
34.     string buf = string.Format("Monster {0} \t
35.     health = {1} \ ammo = {2}", name, health, ammo);
36.    return buf;  }
37.     string name;
38.     int health, ammo;
39.    }
```

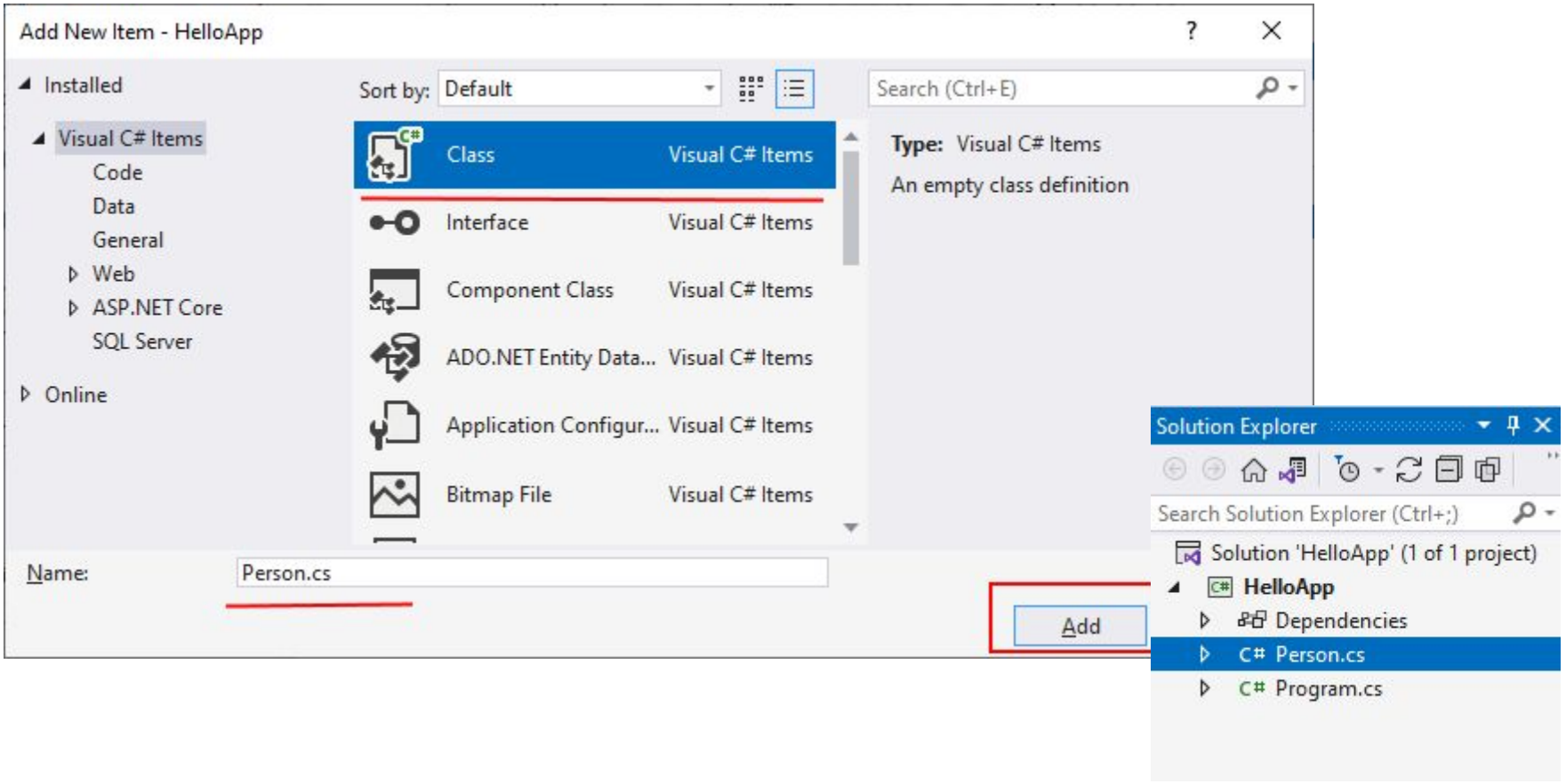
Добавление класса в Visual Studio

Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл. И Visual Studio предоставляет по умолчанию встроенные шаблоны для доба



Добавление класса в Visual Studio

В открывшемся окне добавления нового элемента убедимся, что в центральной части с шаблонами элементов у нас выбран пункт **Class**. А внизу окна в поле **Name** введем название добавляемого класса - пусть он будет называться **Person**:



Добавление класса в Visual Studio

Таким образом, мы можем определять классы в отдельных файлах и использовать их в программе.

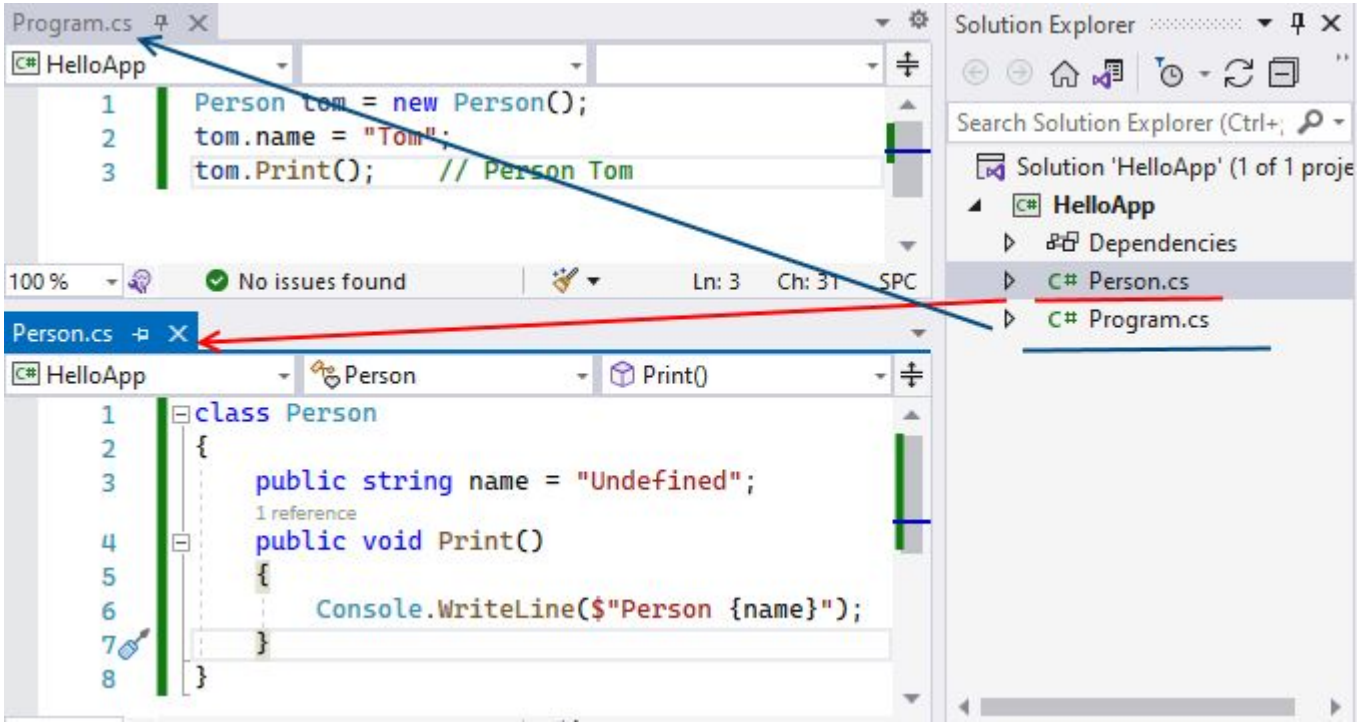
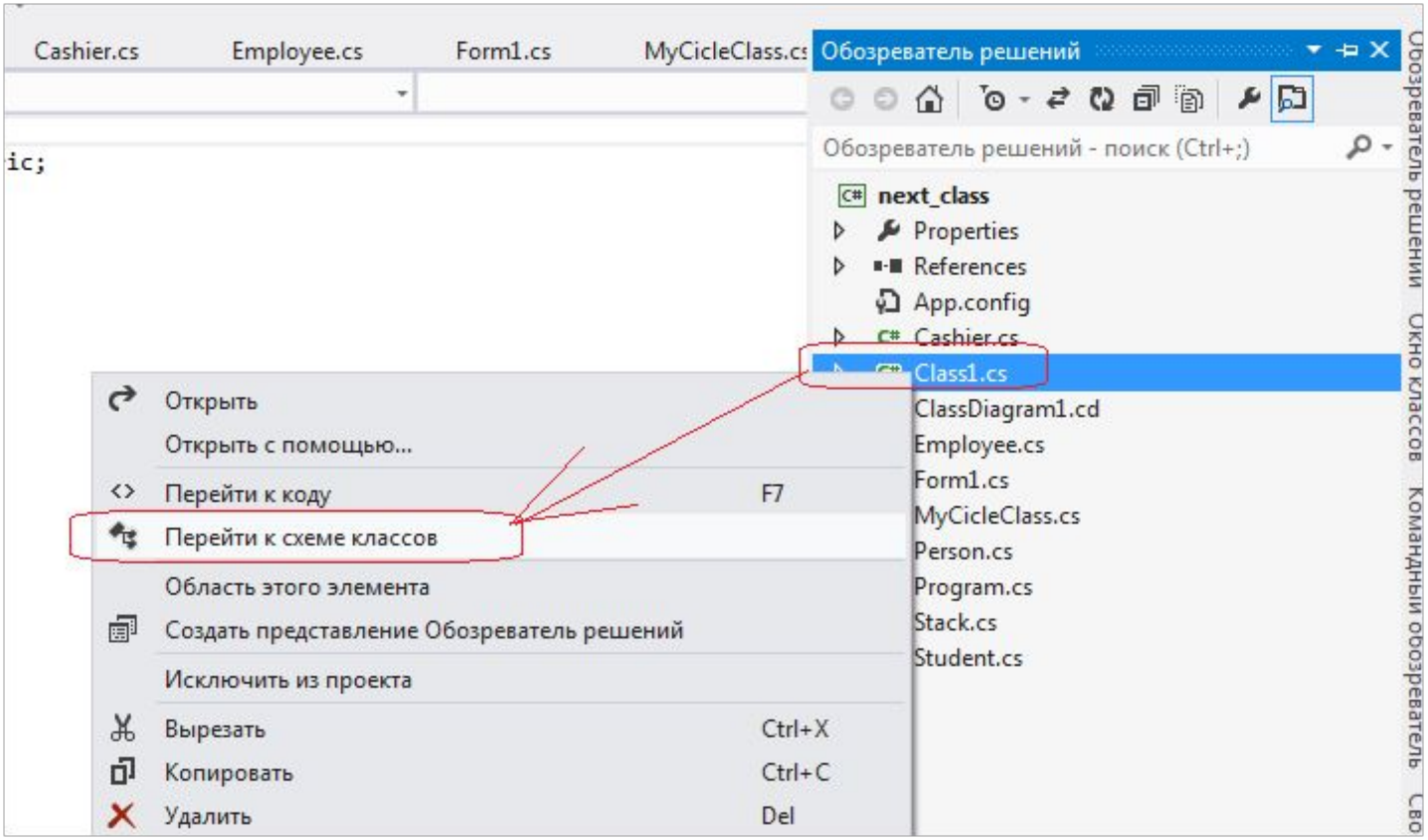


Схема Класса



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();

Demo b = new Demo();
```

- 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 { ... }
```

```
1. class Class1
2.{
3.      static void Main()
4.      {
5.          Monster X = new Monster();
6.          X.Passport();
7.          Monster Vasia = new Monster( "Vasia" );
8.          Vasia.Passport();
9.          Monster Masha = new Monster( 200, 200, "Masha" );
0.          Console.Writeline(Masha);
1.      }
2.  }
```

Результат работы программы:
Monster Noname health = 100 ammo = 100
Monster Vasia health = 100 ammo = 100
Monster Masha health = 200 ammo = 200

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]

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.

Поля класса

- **Поля** служат для хранения данных, содержащихся в объекте. Поля аналогичны переменным, т.к. они непосредственно читаются и устанавливаются.
- **Поле** – это переменная, объявленная внутри класса.
  - Как правило, поля объявляются с модификаторами доступа ***private*** либо ***protected***, чтобы запретить прямой доступ к ним.
  - Для получения доступа к полям следует использовать **свойства** или **методы**.

Пример класса

```
1.  using System;
2.  namespace CA1
3.  {   class Demo
4.     {  public int a = 1;                          // поле данных
5.        public const double  c = 1.66;                  // константа
6.        public static string s = "Demo";                // статическое поле класса
7.        double y;                                 // закрытое поле данных
8.     }
9.     class Class1
10.    {   static void Main()
11.       {
12.          Demo x = new Demo();          // создание экземпляра класса Demo
13.          Console.WriteLine( x.a );     // x.a - обращение к полю класса
14.          Console.WriteLine( Demo.c );  // Demo.c - обращение к константе
15.          Console.WriteLine( Demo.s );  // обращение к статическому полю
16.    }}}
```

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

Конструкторы

Конструктор класса – это специальный метод, который вызывается при инициализации объекта с помощью ключевого слова *new*. Имя конструктора должно совпадать с именем класса.

Свойства конструкторов:

- Конструктор не возвращает значение, даже типа void.
- Конструктор можно не создавать явно, тогда для класса будет создан конструктор по умолчанию.
- 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.
- Если вы создадите конструктор, который содержит набор аргументов, то конструктор по умолчанию уже не будет создан для класса. Если вы хотите создавать объекты без указания аргументов, то необходимо добавить в класс соответствующий конструктор.

Пример класса с конструктором

```
1.    class Demo
2.       {
3.          public Demo( int a, double y )        // конструктор
4.          {
5.             this.a = a;
6.             this.y = y;
7.          }
8.          int a;
9.          double y;
10.      }
11.
12.      class Class1
13.      {   static void Main()
14.          {
15.             Demo a = new Demo( 300, 0.002 );     // вызов конструктора
16.             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
```

Пример класса с несколькими конструкторами

```
class DemoClass
{
 int field = 0;
 public int Property {get;set;}
 public void Method()
  {
    Console.WriteLine("Method");
  }
 public DemoClass()
 {}
 public DemoClass(int field)
 {
   this.field = field;
 }
 public DemoClass(int field, int prop)
 {
   this.field = field;
   Property = prop;
 }
}
```

// Инициализация объектов класса

// Свойство класса

// Метод класса

```
DemoClass demo = new DemoClass();
DemoClass d2 = new DemoClass(1);
d2.Method();  // field: 1, Property: 0
DemoClass d3 = new DemoClass(1, 2);
d3.Method(); // field: 1, Property: 2
DemoClass d5 = new DemoClass(10) { Property
= 11 };
d5.Method(); // field: 10, Property: 11
```

// конструктор класса, содержащий 2 аргумента

Свойства

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

[модификаторы] тип_свойства название_свойства

{

get { действия, выполняемые при получении значения свойства}

set { действия, выполняемые при установке значения свойства}

}

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

- В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.
- В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.
- Блоки **get** и **set** еще называются акссесорами или методами доступа (к значению свойства), а также геттером и сеттером.
- 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).

Поля класса

```
Person person = new Person();
 person.Name = "Tom";
 string personName = person.Name;
Console.WriteLine(personName);   // Tom

class Person
{
    private string name = "Undefined";
    public string Name
    {
        get
        {
            return name;        // возвращаем значение свойства
        }
        set
        {
            name = value;       // устанавливаем новое значение свойства
        }
    }
}
```

Устанавливаем свойство - срабатывает блок Set значение "Tom" – это передаваемое в свойство value

Получаем значение свойства и присваиваем его переменной - срабатывает блок Get

Параметр **value** представляет передаваемое значение, которое передается переменной name.

Свойства

Свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения. Например, нам выполнить проверку по возрасту:

```
Person person = new Person();
 Console.WriteLine(person.Age);  // 1
person.Age = 37;
Console.WriteLine(person.Age);  // 37
person.Age = -23;           // Возраст должен быть в диапазоне от 1 до 120
Console.WriteLine(person.Age);  // 37 - возраст не изменился

class Person
{
   int age = 1;
   public int Age
   {
      set
      {
         if (value < 1 || value > 120)
            Console.WriteLine("Возраст должен быть в диапазоне от 1 до 120");
         else
            age = value;
      }
      get { return age; }
   }
}
```

изменяем значение свойства

пробуем передать недопустимое значение

```
1
37
Возраст должен быть в диапазоне от 1 до 120
37
```

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

Сквозной пример класса

```
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 GetAmmo()
    {  return ammo; }
```

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

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

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