

ЛК 8. Отношения между классами

Отношения между классами и объектами

Отношения между классами и объектами

В ООП выделяют 5 вариантов отношений между объектами:

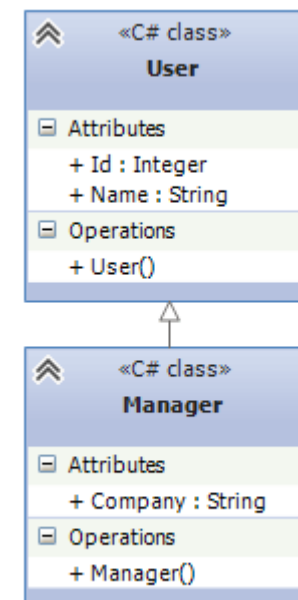
- наследование,
- реализация,
- ассоциация,
- композиция
- агрегация.

Наследование

- Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского).
- Нередко отношения наследования еще называют генерализацией или обобщением.
- Наследование определяет отношение **IS A**, то есть "является".

```
class User
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
}

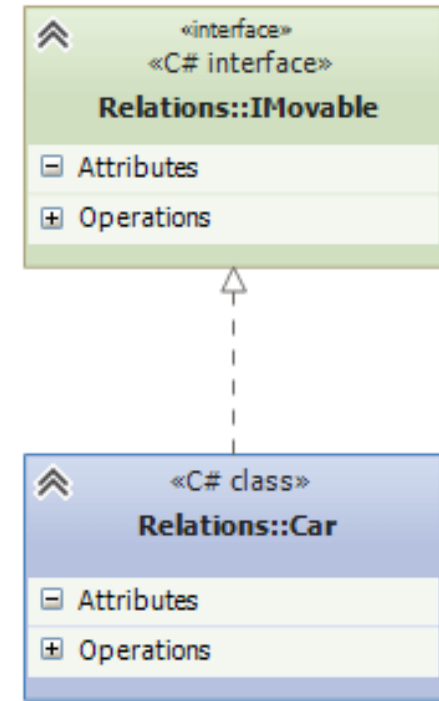
0 references
class Manager : User
{
    0 references
    public string Company { get; set; }
}
```



Реализация

- Реализация предполагает определение интерфейса и его реализация в классах.
- Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

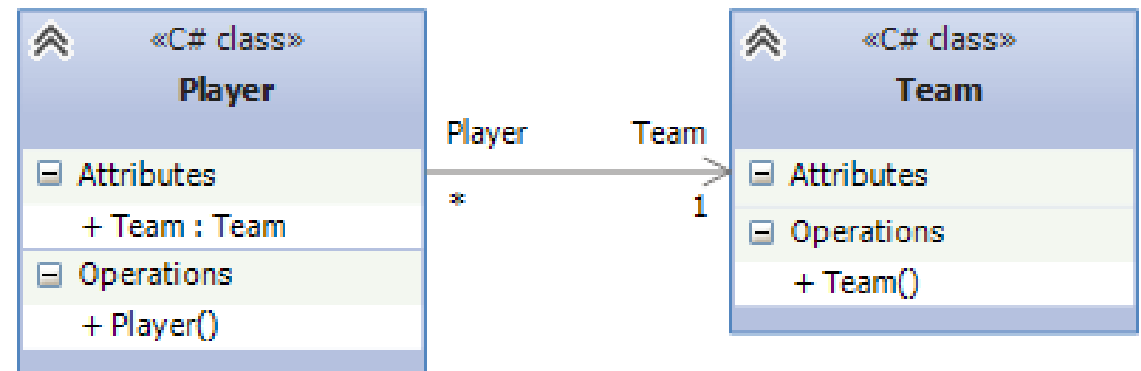
```
public interface IMovable
{
    1 reference
    void Move();
}
0 references
public class Car : IMovable
{
    1 reference
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```



Ассоциация

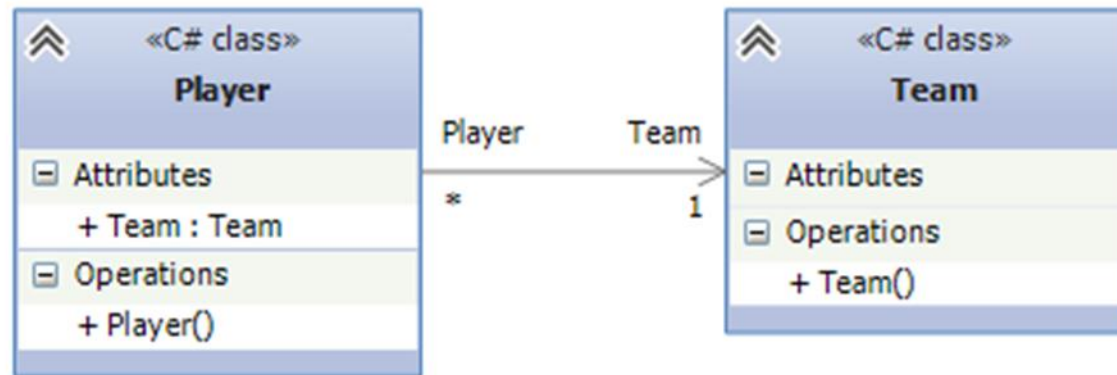
- Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа.
- Например, объект одного типа содержит или использует объект другого типа.
- Например, игрок играет в определенной команде:

```
class Team
{
}
References
class Player
{
    References
    public Team Team { get; set; }
}
```



Ассоциация

- Нередко при отношении ассоциации указывается кратность связей.
- В данном случае единица у **Team** и звездочка у **Player** на диаграмме отражает связь **1-ко-многим**.
- То есть одна команда будет соответствовать многим игрокам.
- **Агрегация** и **композиция** являются частными случаями ассоциации.

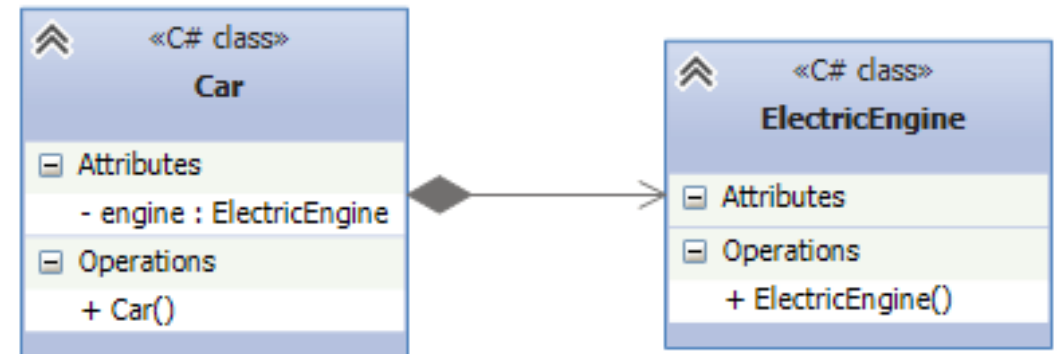


Композиция

- Композиция определяет отношение **HAS A**, то есть отношение "имеет".
- Например, класс автомобиля содержит объект класса двигателя:

```
public class Engine
{ }

1 reference
public class Car
{
    Engine engine;
    0 references
    public Car()
    {
        engine = new Engine();
    }
}
```

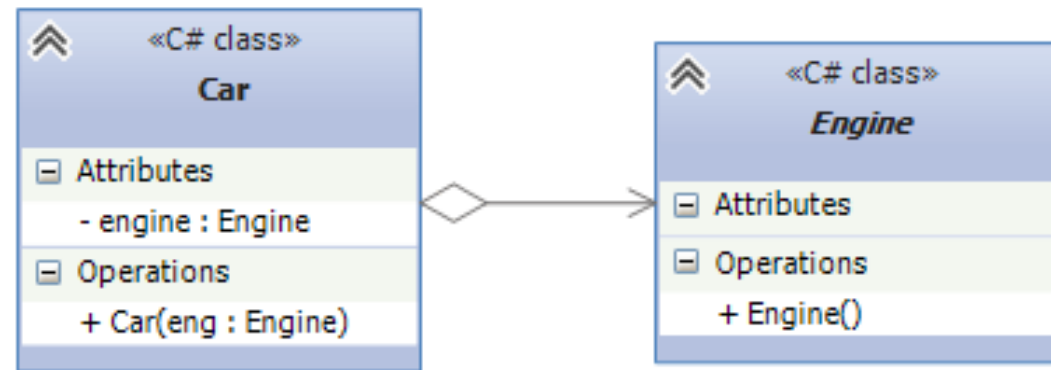


Агрегация

- От композиции следует отличать агрегацию.
- Она также предполагает отношение **HAS A**, но реализуется она иначе:

```
public class Engine
{ }

1 reference
public class Car
{
    Engine engine;
    0 references
    public Car(Engine eng)
    {
        engine = eng;
    }
}
```

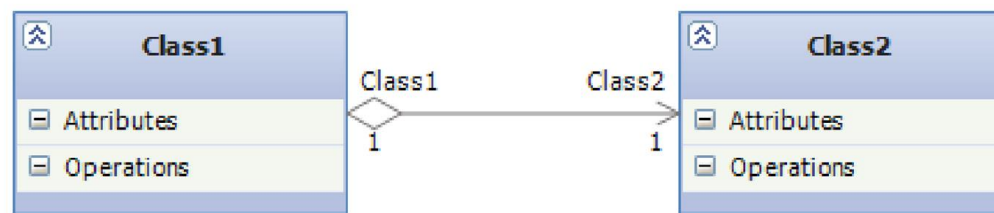


Агрегация и композиция

Агрегация

Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют *агрегацию по ссылке*, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбиком на блоке класса и линией, идущей от этого ромбика к содержащемуся классу.

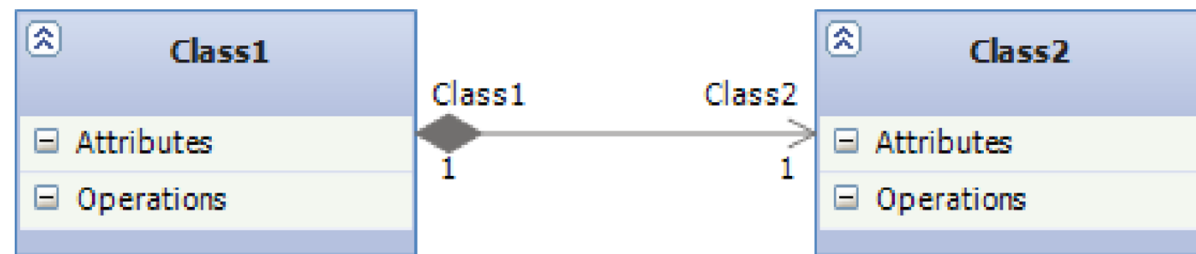


КОМПОЗИЦИЯ

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком.



Различия между композицией и агрегацией

Комната является неотделимой частью квартиры, следовательно здесь подходит *композиция* потому что комната без квартиры существовать не может.



Различия между композицией и агрегацией

А, например, мебель не является неотъемлемой частью квартиры, но в то же время, квартира содержит мебель, поэтому следует использовать *агрегацию*.



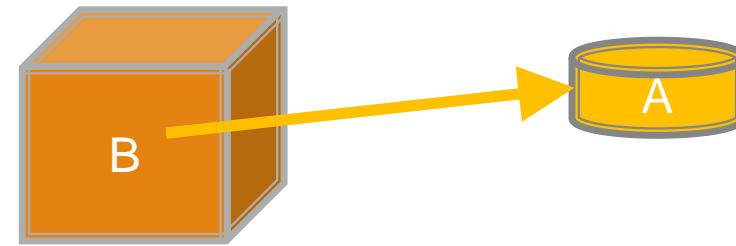
Агрегация и композиция

Допустим, существует некий класс A

```
class A  
{  
    ...  
}
```

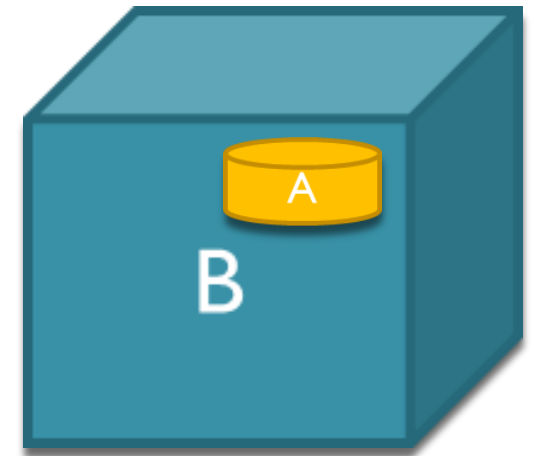
Агрегация

```
class B
{
    private A _a;
    public B(A a)
    // Объект A живет где-то отдельно
    // (суть не в конструкторе)
    {
        _a = a;
    }
}
```



Композиция

```
class B
{
    private A _a = new A();
    // Объект A существует только вместе с B
}
```

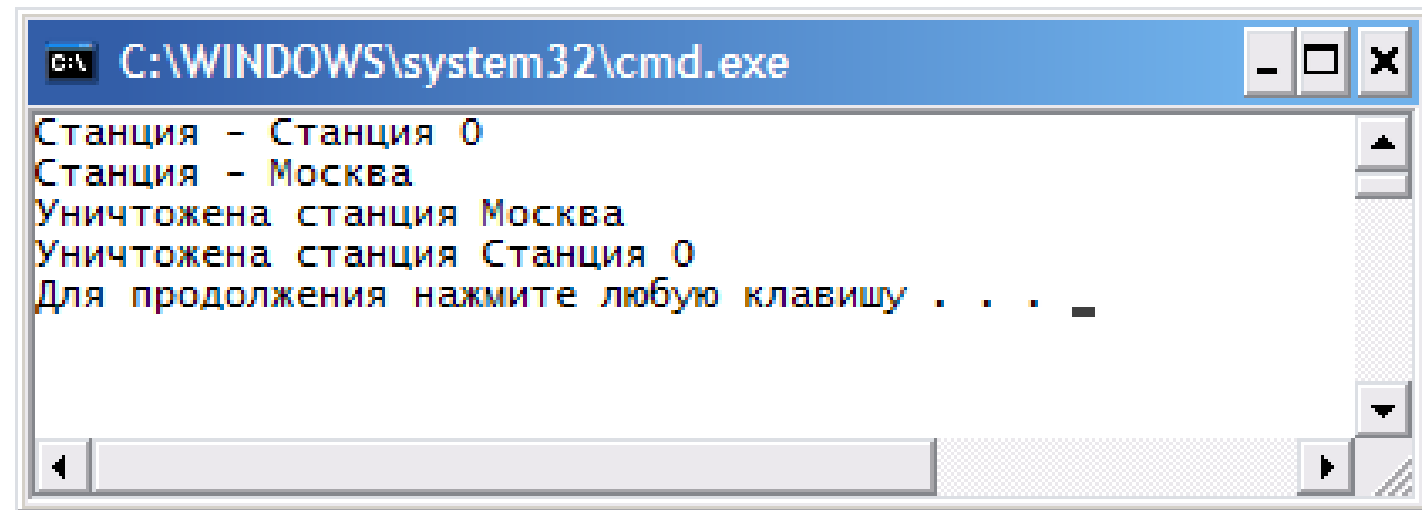


Класс Station

```
class Station
{
    public string name; // название станции
    public Station()
    {
        name = "Станция 0";
    }
    public Station(string name)
    {
        this.name = name;
    }
    public void Print()
    {
        Console.WriteLine("Станция - " + name);
    }
    ~Station()
    {
        Console.WriteLine("Уничтожена станция " + name);
    }
}
```

Класс Station

```
Station s1 = new Station();  
Station s2 = new Station("Москва");  
s1.Print();  
s2.Print();
```

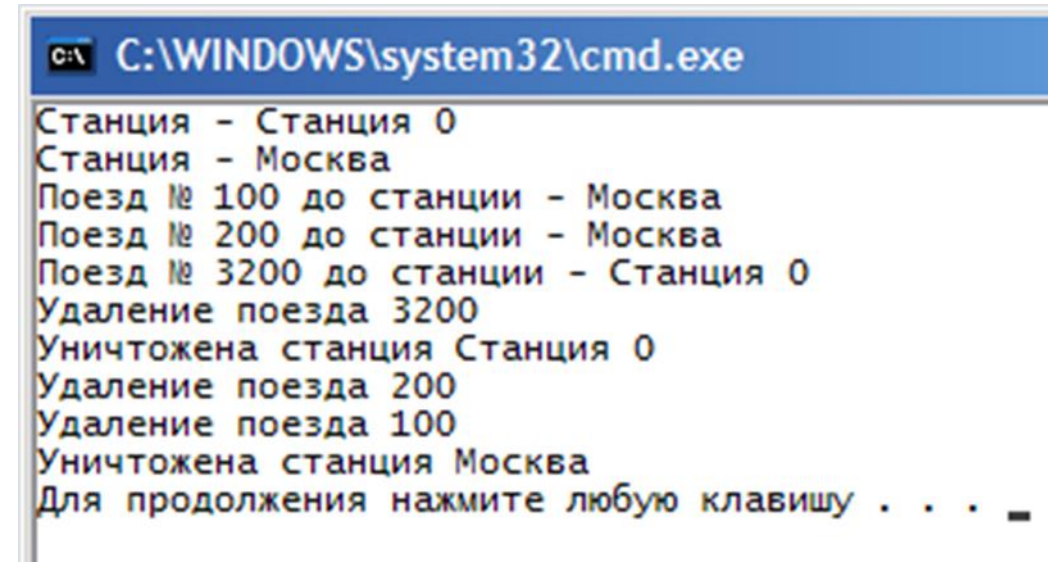


Класс Train

```
class Train
{
    public int n;           //номер поезда
    public Station st;      // станция назначения
    public Train(){}
    public Train(int n, Station st)
    {
        this.n = n;
        this.st = st;
    }
    public void Print()
    {
        Console.WriteLine("Поезд № " + n + " до станции - " + st.name);
    }
    ~Train()
    {
        Console.WriteLine("Удаление поезда " + n);
    }
}
```

Агрегация

```
static void Main(string[] args)
{
    Station s1 = new Station();
    Station s2 = new Station("Москва");
    s1.Print();
    s2.Print();
    Train tr1 = new Train(100,s2);
    tr1.Print();
    Train tr2 = new Train(200, s2);
    tr2.Print();
    Train tr3 = new Train(3200, s1);
    tr3.Print();
}
```



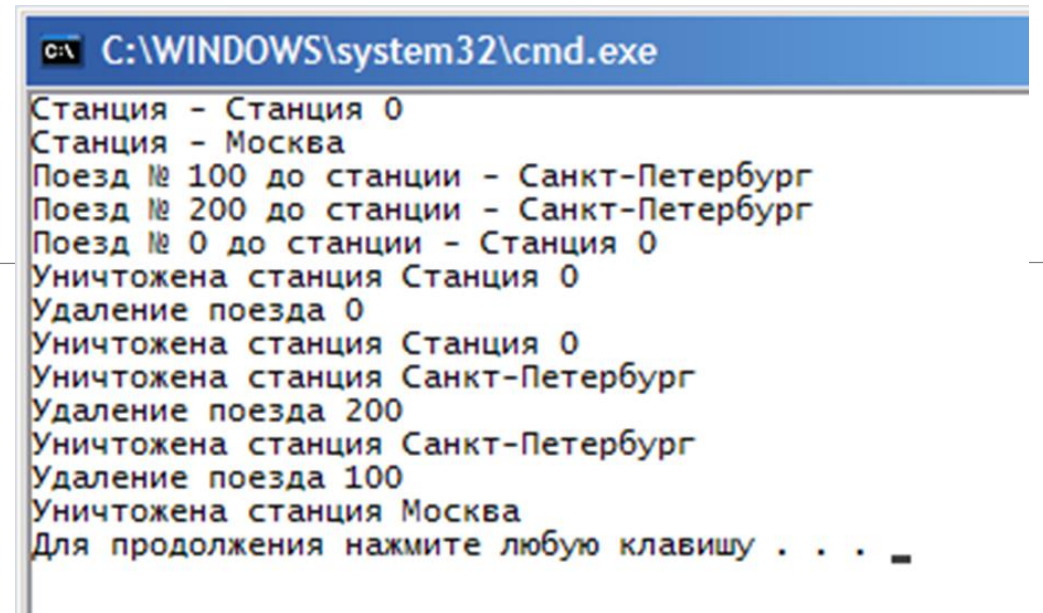
```
C:\WINDOWS\system32\cmd.exe
Станция - Станция 0
Станция - Москва
Поезд № 100 до станции - Москва
Поезд № 200 до станции - Москва
Поезд № 3200 до станции - Станция 0
Удаление поезда 3200
Уничтожена станция Станция 0
Удаление поезда 200
Удаление поезда 100
Уничтожена станция Москва
Для продолжения нажмите любую клавишу . . . _
```

Композиция

```
class Train
{
    public int n;//номер поезда
    public Station st;// станция назначения
    public Train()
    {
        n = 0;
        st = new Station("Станция 0");
    }
    public Train(int n, string str)
    {
        this.n = n;
        this.st = new Station(str);
    }
    public void Print()
    {
        Console.WriteLine("Поезд № " + n + " до станции - " + st.name);
    }
}
```

КОМПОЗИЦИЯ

```
static void Main(string[] args)
{
    Station s1 = new Station();
    Station s2 = new Station("Москва");
    s1.Print();
    s2.Print();
    Train tr1 = new Train(100, "Санкт-Петербург");
    tr1.Print();
    Train tr2 = new Train(200, "Санкт-Петербург");
    tr2.Print();
    Train tr3 = new Train();
    tr3.Print();
}
```



```
C:\WINDOWS\system32\cmd.exe
Станция - Станция 0
Станция - Москва
Поезд № 100 до станции - Санкт-Петербург
Поезд № 200 до станции - Санкт-Петербург
Поезд № 0 до станции - Станция 0
Уничтожена станция Станция 0
Удаление поезда 0
Уничтожена станция Станция 0
Уничтожена станция Санкт-Петербург
Удаление поезда 200
Уничтожена станция Санкт-Петербург
Удаление поезда 100
Уничтожена станция Москва
Для продолжения нажмите любую клавишу . . . _
```

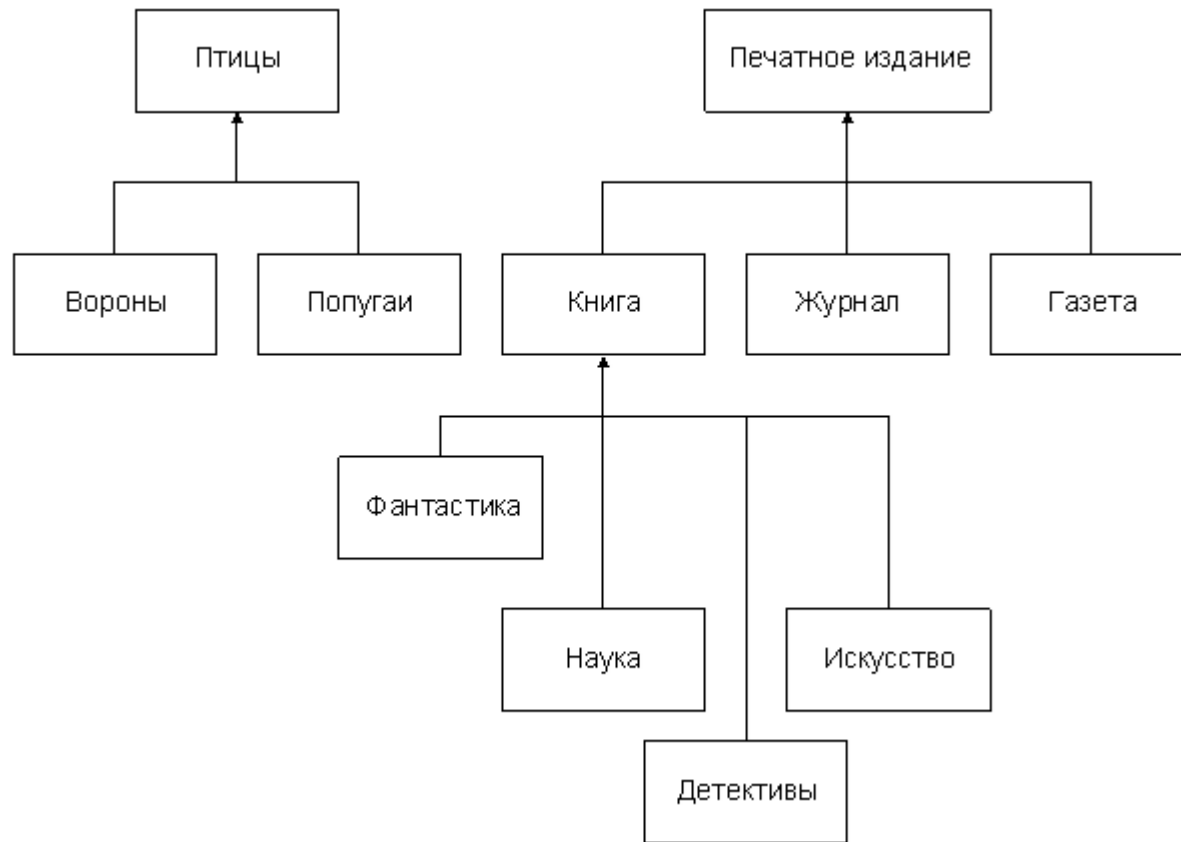


Наследование

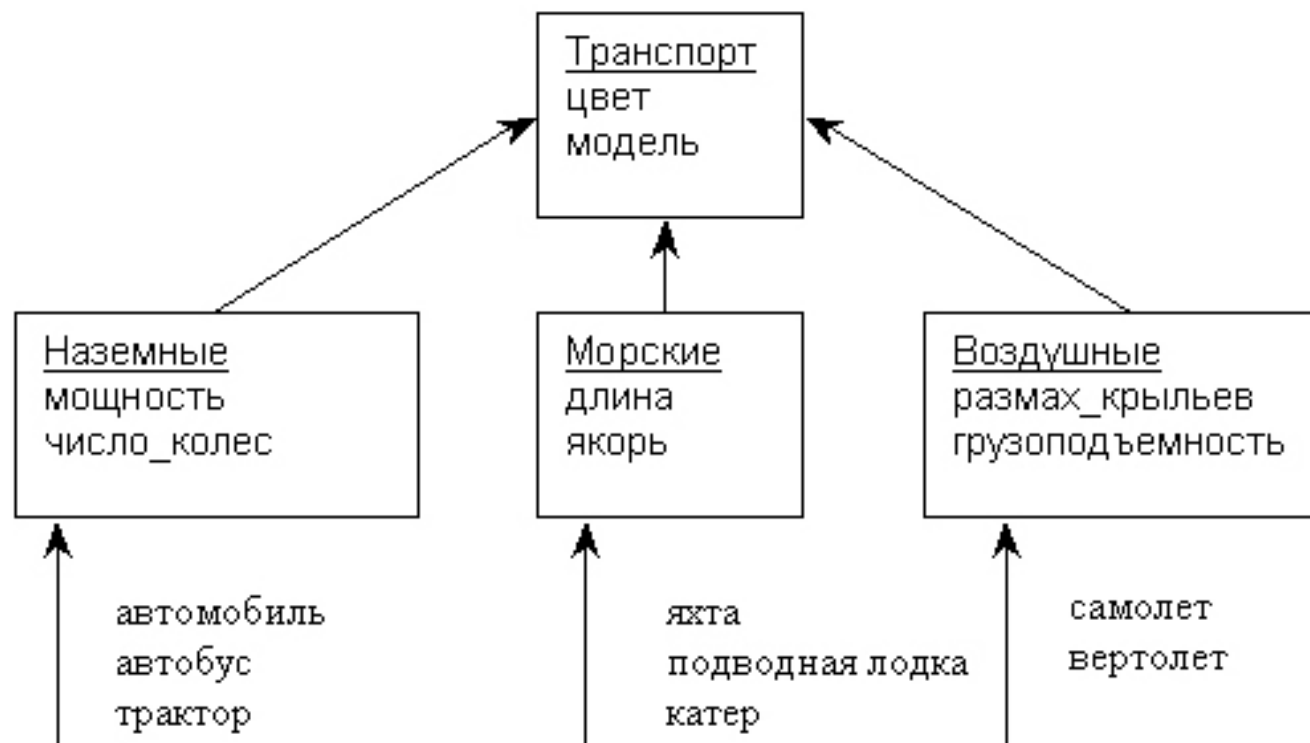
Наследование

Наслédование — механизм объектно-ориентированного программирования, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

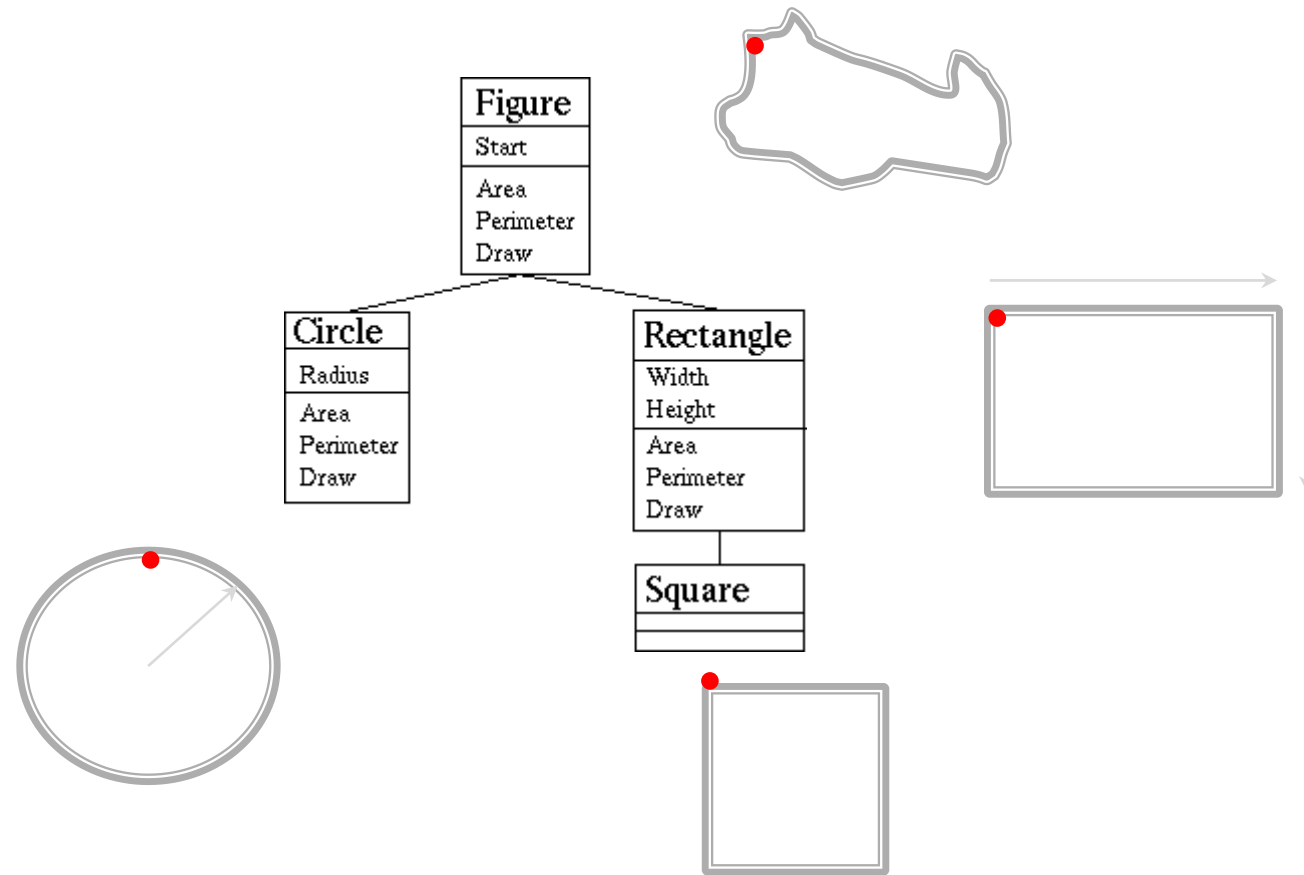
Наследование



Наследование



Наследование

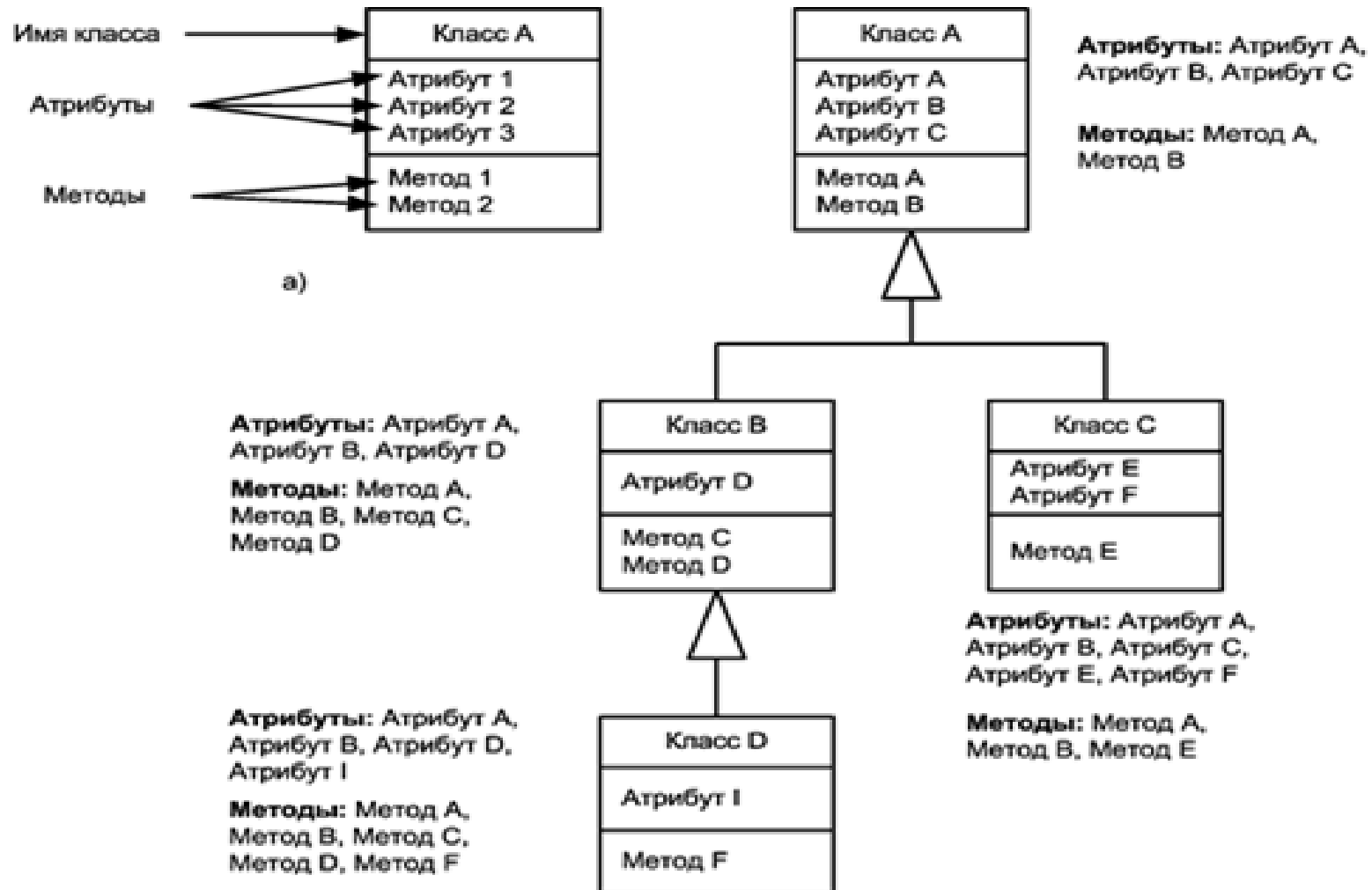


Наследование

Класс, от которого произошло наследование, называется базовым или родительским (base class).

Классы, которые произошли от базового, называются потомками, наследниками, подклассами или производными классами (derived class).

Наследование



Наследование

При наследовании все атрибуты и методы родительского класса наследуются классом-потомком.

Наследование может быть многоуровневым, и тогда классы, находящиеся на нижних уровнях иерархии, унаследуют все свойства (атрибуты и методы) всех классов, прямыми или косвенными потомками которых они являются.

Класс В унаследует атрибуты и методы класса А и, следовательно, будет обладать атрибутами А, В, С и D и методами А, В, С и D, а класс С – атрибутами А, В, С, Е, F и методами А, В и Е.

Наследование

Помимо единичного, существует и множественное наследование, когда класс наследует сразу нескольким классам.

При этом он унаследует свойства всех классов, потомком которых он является.

При использовании множественного наследования необходимо быть особенно внимательным, так как возможны коллизии, когда класс-потомок может унаследовать одноименные свойства, с различным содержанием.

C# не поддерживает множественное наследование.

Полиморфизм

При наследовании одни методы класса могут замещаться другими.

Так, класс транспортных средств будет обладать обобщенным методом движения.

В классах-потомках этот метод будет конкретизирован: автомобиль будет ездить, самолет – летать, корабль – плавать.

Такое изменение семантики метода называется полиморфизмом.

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию. (Один интерфейс, множество реализаций)

Наследование

```
class имя_наследника : имя_базового_класса  
{тело класса}
```

Наследник обладает всеми полями, методами и свойствами предка, однако элементы предка с модификатором `private` не доступны в наследнике.

При наследовании нельзя расширить область видимости класса:

`internal`—класс может наследоваться от `public`—класса, но не наоборот

Наследование

Ничего не делающий самостоятельно потомок не эффективен, от него мало проку. Что же может делать потомок?

Прежде всего, он может добавить новые свойства - поля класса.

Заметьте, потомок не может ни отменить, ни изменить модификаторы или типы полей, наследованных от родителя - он может только добавить собственные поля.

класс "Питомец"

```
class Pet
{
    public int eyes = 2;    //глаза
    public int tail;        // хвост
    public int legs;        // ноги
    0 references
    public Pet()
    {
        tail = 1;
    }
    0 references
    public void Speak()
    {
        Console.WriteLine("I'm a pet");
    }
}
```

Это общий класс.

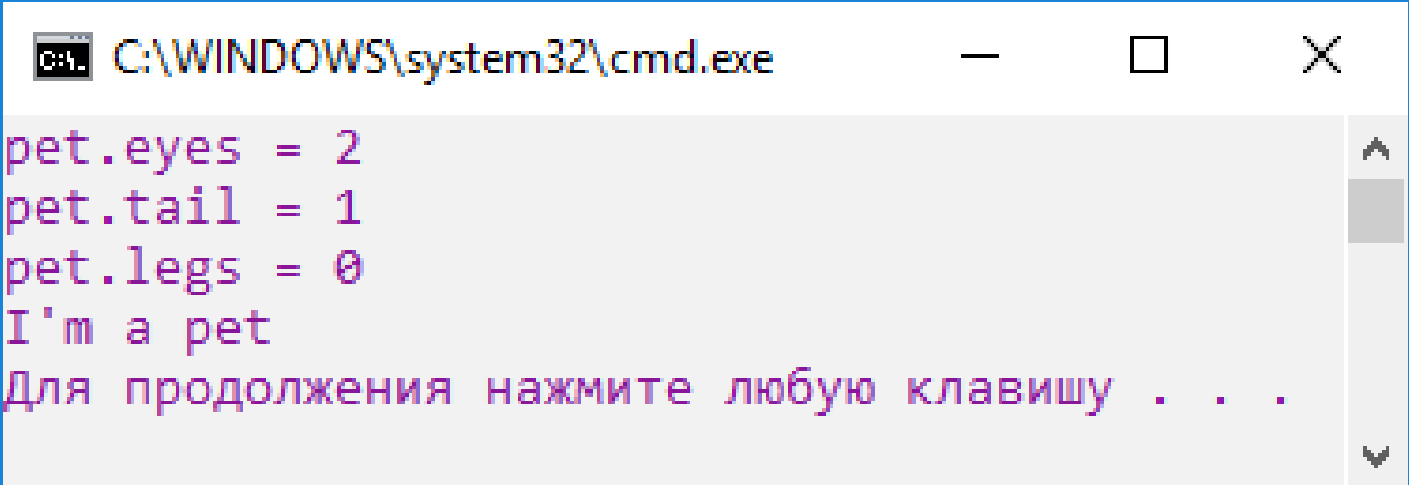
В нем могут быть все
виды питомцев
(собаки, птички,
рыбки).

У всех 2 глаза, 1 хвост и
какое-то к-во ног.

Все могут говорить.

Наследование

```
static void Main(string[] args)
{
    Pet pet = new Pet();
    Console.WriteLine("pet.eyes = " + pet.eyes);
    Console.WriteLine("pet.tail = " + pet.tail);
    Console.WriteLine("pet.legs = " + pet.legs);
    pet.Speak();
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the output of a C# program. The output consists of four lines of text in a purple monospace font: "pet.eyes = 2", "pet.tail = 1", "pet.legs = 0", and "I'm a pet". Below these lines, there is a prompt "Для продолжения нажмите любую клавишу . . ." (Press any key to continue). The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

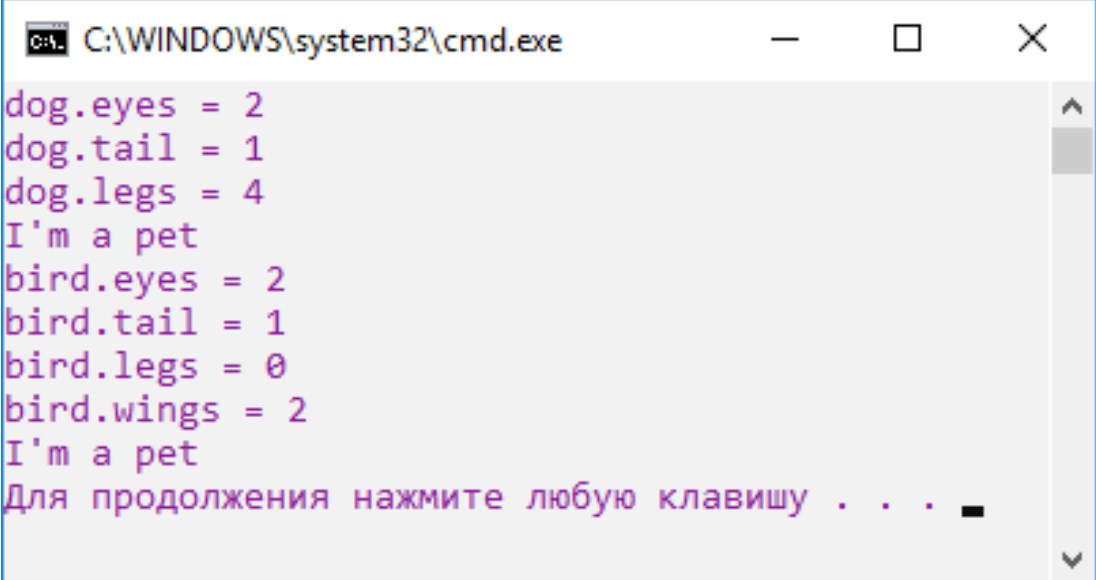
```
C:\WINDOWS\system32\cmd.exe
pet.eyes = 2
pet.tail = 1
pet.legs = 0
I'm a pet
Для продолжения нажмите любую клавишу . . .
```

Создадим наследников

```
class Dog : Pet
{
    0 references
    public Dog()
    {
        legs = 4;
    }
}
0 references
class Bird : Pet
{
    public int wings = 2;
}
```

Наследование

```
Dog dog = new Dog();
Console.WriteLine("dog.eyes = " + dog.eyes);
Console.WriteLine("dog.tail = " + dog.tail);
Console.WriteLine("dog.legs = " + dog.legs);
dog.Speak();
Bird bird = new Bird();
Console.WriteLine("bird.eyes = " + bird.eyes);
Console.WriteLine("bird.tail = " + bird.tail);
Console.WriteLine("bird.legs = " + bird.legs);
Console.WriteLine("bird.wings = " + bird.wings);
bird.Speak();
```



```
C:\WINDOWS\system32\cmd.exe
dog.eyes = 2
dog.tail = 1
dog.legs = 4
I'm a pet
bird.eyes = 2
bird.tail = 1
bird.legs = 0
bird.wings = 2
I'm a pet
Для продолжения нажмите любую клавишу . . .
```

Собаки имеют свой конструктор с к-вом ног = 4. У птиц появилось новое поле – крылья.

Все пока говорят одно и тоже.

Обратите внимание, что родительский конструктор сработал во всех случаях (это хвост).

Наследование

Все члены родительского класса **public**.

Давайте попробуем сделать скрытыми глаза:

```
class Pet
{
    int eyes = 2; //глаза
    public int tail; // хвост
    public int legs; // ноги
    0 references
    public Pet()
    {
```

Наследование

```
Dog dog = new Dog();  
Console.WriteLine("dog.eyes = " + dog.eyes);  
Console.WriteLine("dog.tail = " + dog.tail);  
  
Bird bird = new Bird();  
Console.WriteLine("bird.eyes = " + bird.eyes);  
Console.WriteLine("bird.tail = " + bird.tail);
```

То, что появятся ошибки при выводе, понятно.

Но как быть с наследниками?

Наследники должны видеть члены родительского класса.

Наследование

Давайте заведем одноглазую собаку.

```
class Dog : Pet
{
    1 reference
    public Dog()
    {
        legs = 4;
        eyes = 1;
    }
}
```

private скрывает и от наследников.

Наследование

Напомню, хорошей стратегией является стратегия "ничего не скрывать от потомков".

Какой родитель знает, что именно из сделанного им может понадобиться потомкам?

Для этого существует `protected`.

Наследование

```
class Pet
{
    protected int eyes = 2;    //глаза
    public int tail;           // хвост
    public int legs;           // ноги
    0 references
```

```
Dog dog = new Dog();
Console.WriteLine("dog.eyes = " + dog.eyes);
Console.WriteLine("dog.tail = " + dog.tail);
```

```
class Dog : Pet
{
    1 reference
    public Dog()
    {
        legs = 4;
        eyes = 1;
    }
```

Наследование

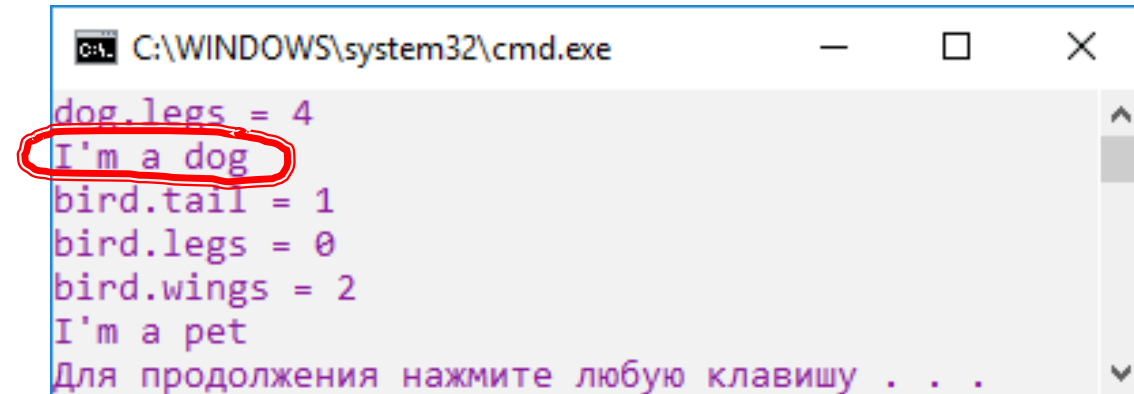
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим.
- То есть, если базовый класс у нас имеет тип доступа `internal`, то производный класс может иметь тип доступа `internal` или `private`, но не `public`.

Добавим свои слова собаке

```
class Dog : Pet
{
    1 reference
    public Dog()
    {
        legs = 4;
        eyes = 1;
    }
    1 reference
    public void Speak()
    {
        Console.WriteLine("I'm a dog");
    }
}
```

Наследование

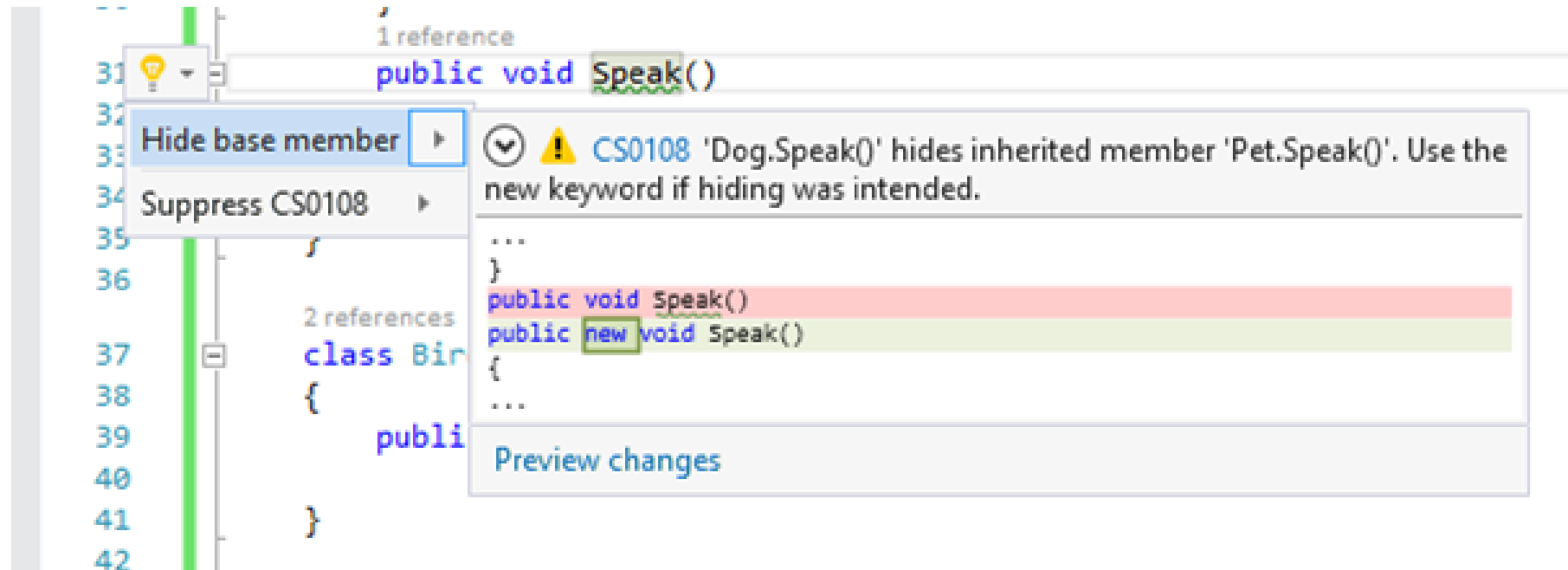
```
Dog dog = new Dog();  
// Console.WriteLine("dog.eyes = " + dog.eyes);  
Console.WriteLine("dog.tail = " + dog.tail);  
Console.WriteLine("dog.legs = " + dog.legs);  
dog.Speak();  
Bird bird = new Bird();  
// Console.WriteLine("bird.eyes = " + bird.eyes);  
Console.WriteLine("bird.tail = " + bird.tail);  
Console.WriteLine("bird.legs = " + bird.legs);  
Console.WriteLine("bird.wings = " + bird.wings);  
bird.Speak();
```



```
C:\WINDOWS\system32\cmd.exe  
dog.legs = 4  
I'm a dog  
bird.tail = 1  
bird.legs = 0  
bird.wings = 2  
I'm a pet  
Для продолжения нажмите любую клавишу . . .
```


Наследование

Но такая грубая перегрузка метода вызовет предупреждение.



Наследование

Для явного указания компилятору, что мы знаем, что делаем, используется слово **new**.

```
class Dog : Pet
{
    !reference
    public Dog()
    {
        legs = 4;
        eyes = 1;
    }
    !reference
    new public void Speak()
    {
        Console.WriteLine("I'm a dog");
    }
}
```

Такой способ называется ***скрытием метода***.

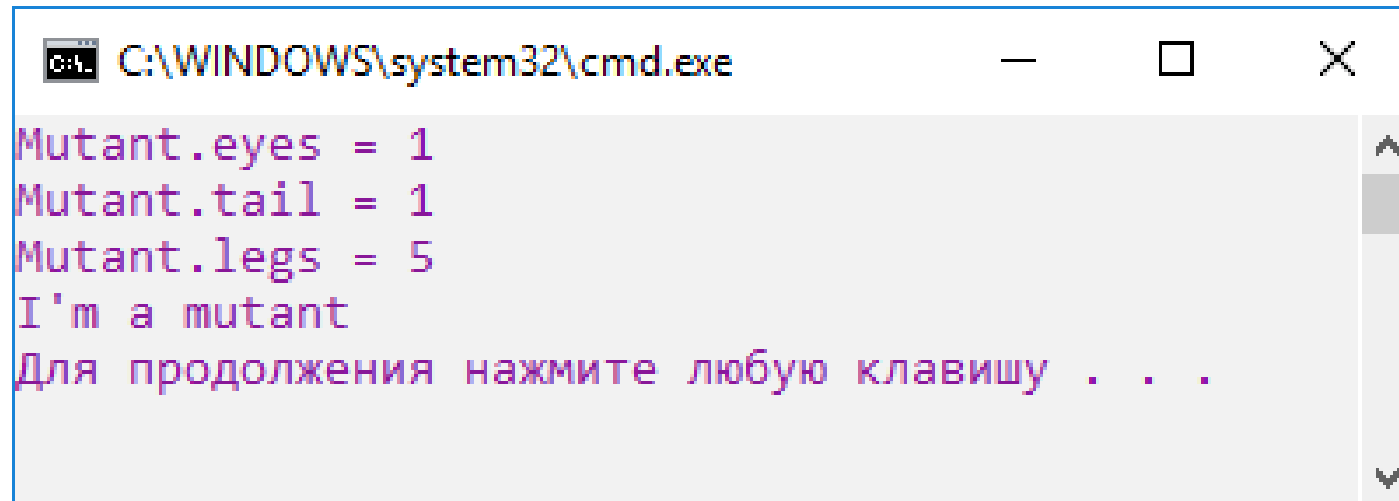
Наследование

Класс-наследник может иметь собственных наследников.

```
class Mutant : Dog
{
    Oreferences
    public Mutant()
    {
        legs = 5;
    }
    Oreferences
    new public void Speak()
    {
        Console.WriteLine("I'm a mutant");
    }
}
```

Наследование

```
Mutant mutant = new Mutant();  
Console.WriteLine("Mutant.eyes = " + mutant.eyes);  
Console.WriteLine("Mutant.tail = " + mutant.tail);  
Console.WriteLine("Mutant.legs = " + mutant.legs);  
mutant.Speak();
```



A screenshot of a Windows command prompt window. The title bar shows the path "C:\WINDOWS\system32\cmd.exe" and standard window controls. The command prompt displays the output of the C# code: "Mutant.eyes = 1", "Mutant.tail = 1", "Mutant.legs = 5", and "I'm a mutant". Below this, it shows a prompt "Для продолжения нажмите любую клавишу . . ." (Press any key to continue). The text is displayed in a purple font on a light gray background.

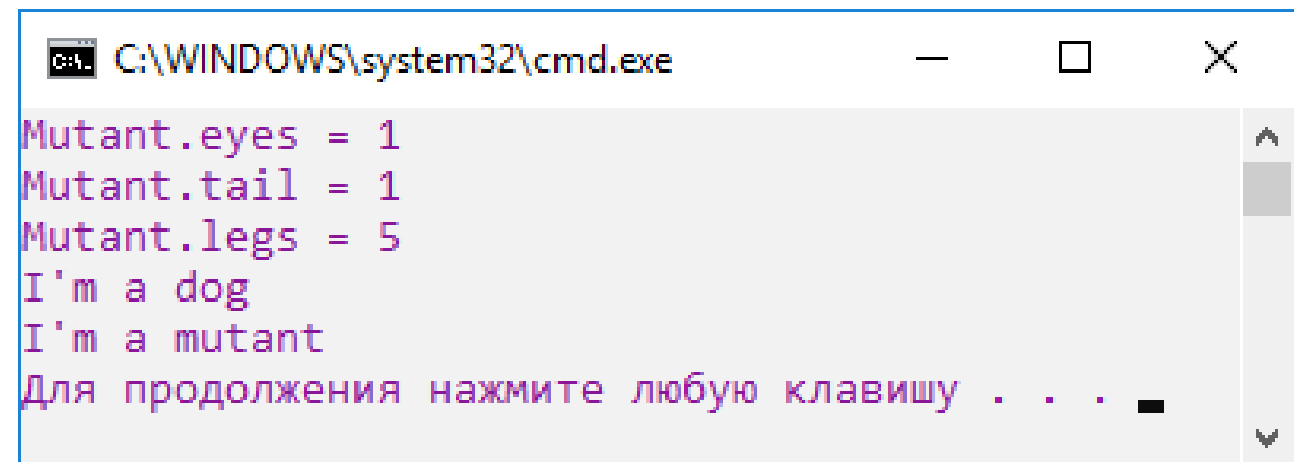
```
C:\WINDOWS\system32\cmd.exe  
Mutant.eyes = 1  
Mutant.tail = 1  
Mutant.legs = 5  
I'm a mutant  
Для продолжения нажмите любую клавишу . . .
```

Наследование

Но при этом она еще помнит, что она собака.

Метод **Speak** вызывает метод **Speak** родительского класса (который был скрыт):

```
class Mutant : Dog
{
    1 reference
    public Mutant()
    {
        legs = 5;
    }
    1 reference
    new public void Speak()
    {
        base.Speak();|
        Console.WriteLine("I'm a mutant");
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
Mutant.eyes = 1
Mutant.tail = 1
Mutant.legs = 5
I'm a dog
I'm a mutant
Для продолжения нажмите любую клавишу . . .
```

Наследование

Для вызова метода родительского класса используется ключевое слово **base**.

Кстати, для обращения к собственному объекту, если это необходимо, используется слово **this**.

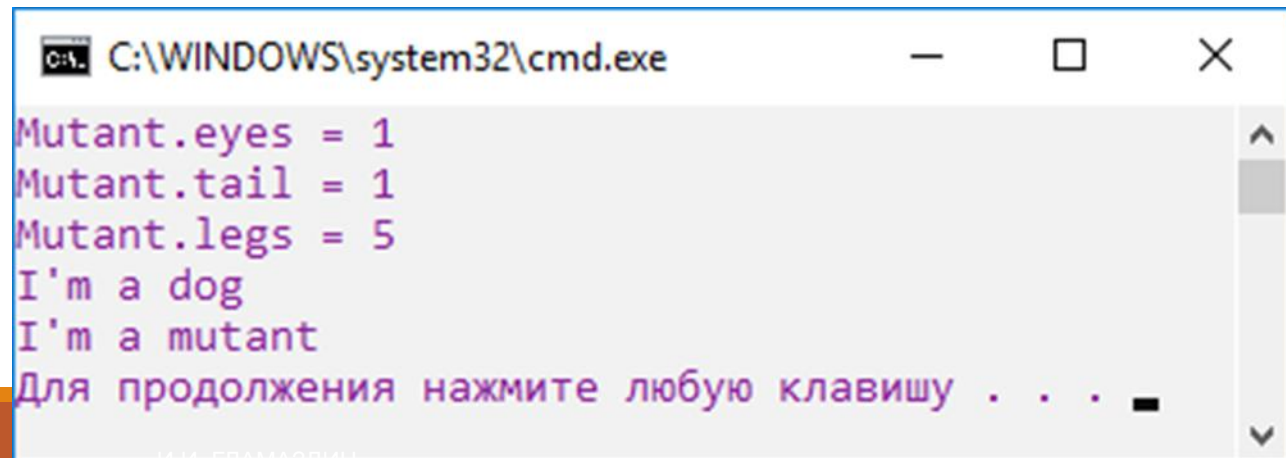
Наследование

Обратите внимание, как здесь сработали конструкторы: сначала был вызван конструктор самого первого предка (**Pet**). Один хвост задавался именно там. Потом сработал конструктор его потомка: **Dog**. 4 ноги и 1 глаз. И уже в последнюю очередь – собственный конструктор **Mutant** – ног стало 5.

```
public Pet()
{
    tail = 1;
}
```

```
public Dog()
{
    legs = 4;
    eyes = 1;
}
```

```
public Mutant()
{
    legs = 5;
}
```



```
C:\WINDOWS\system32\cmd.exe
Mutant.eyes = 1
Mutant.tail = 1
Mutant.legs = 5
I'm a dog
I'm a mutant
Для продолжения нажмите любую клавишу . . .
```

Конструкторы при наследовании

Конструкторы не передаются производному классу при наследовании. И если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами, то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base**.

Либо можно определить в базовом классе конструктор без параметров, и тогда он будет вызываться в любом конструкторе наследника, где нет явного вызова родительского конструктора.

Класс "Рыбка"

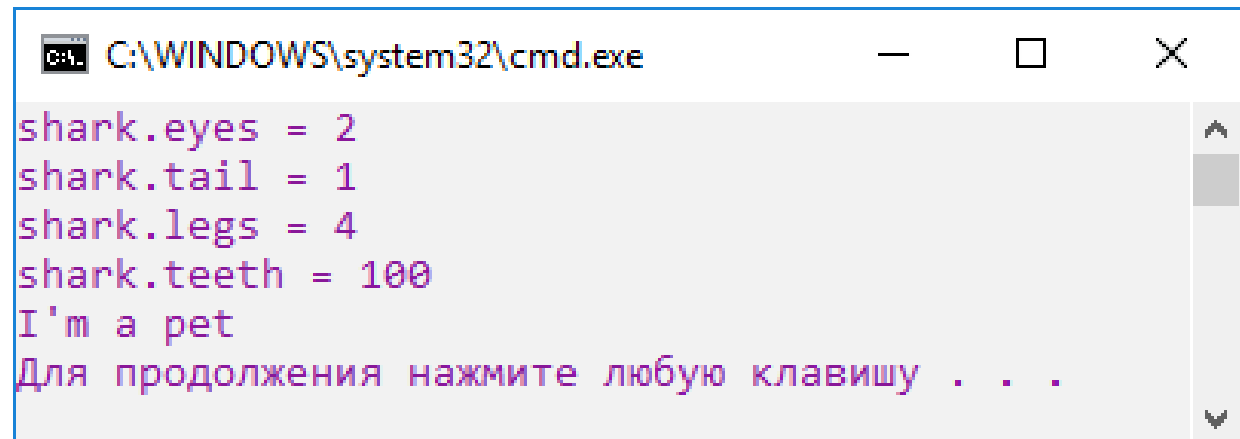
```
class Fish : Pet
{
    public int fins; // плавники
    0 references
    public Fish(int eyes, int tail, int fins)
    {
        this.eyes = eyes;
        this.tail = tail;
        this.fins = fins;
    }
}
```

Класс "Акула"

```
-----  
class Shark : Fish  
{  
    public int teeth;    // зубы  
    0 references  
    public Shark(int eyes, int tail, int fins, int teeth)  
        : base(eyes, tail, fins)  
    {  
        this.teeth = teeth;  
    }  
}
```

Наследование

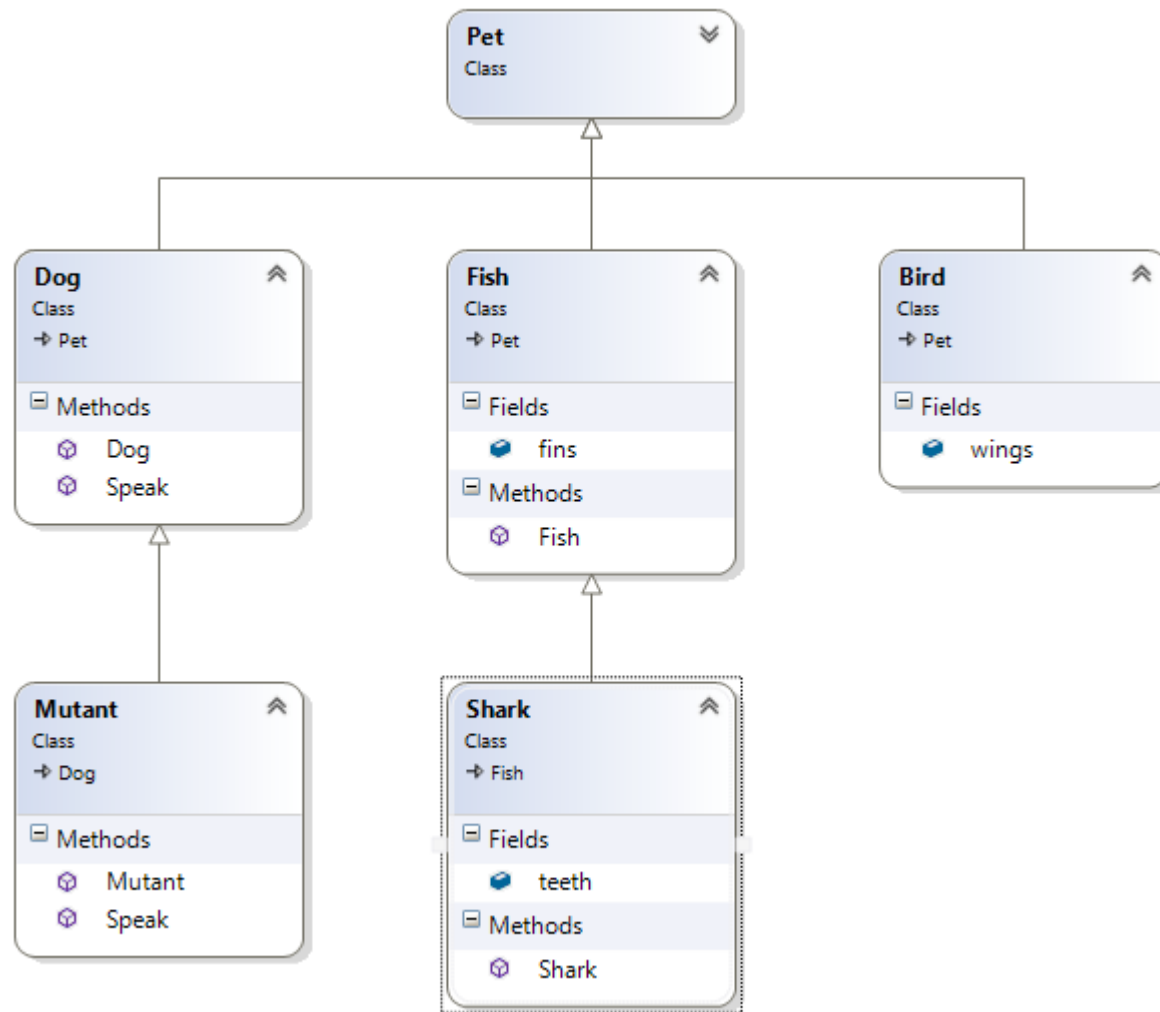
```
Shark shark = new Shark(2, 1, 4, 100);  
Console.WriteLine("shark.eyes = " + shark.eyes);  
Console.WriteLine("shark.tail = " + shark.tail);  
Console.WriteLine("shark.legs = " + shark.fins);  
Console.WriteLine("shark.teeth = " + shark.teeth);  
shark.Speak();
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the output of the C# code: "shark.eyes = 2", "shark.tail = 1", "shark.legs = 4", "shark.teeth = 100", and "I'm a pet". The prompt "Для продолжения нажмите любую клавишу . . ." is visible at the bottom.

```
C:\WINDOWS\system32\cmd.exe  
shark.eyes = 2  
shark.tail = 1  
shark.legs = 4  
shark.teeth = 100  
I'm a pet  
Для продолжения нажмите любую клавишу . . .
```

Диаграмма классов





Отношение между классами

ПРЕОБРАЗОВАНИЕ ТИПОВ

Преобразование типов

Код С# является строго типизированным во время компиляции.

Это значит, что после объявления переменной ее нельзя объявить повторно или назначить ей значения другого типа, если этот тип невозможно неявно преобразовать в тип переменной.

Например, `string` невозможно неявно преобразовать в `int`.

Преобразование типов

Если переменной одного типа нужно присвоить значение другого типа, то такого рода операции называются ***преобразованиями типа***.

Преобразование типов

Неявные преобразования.

Специальный синтаксис не требуется, так как преобразование всегда завершается успешно и данные не будут потеряны.

Пример - преобразования из меньших в большие целочисленные типы и преобразования из производных классов в базовые классы.

Преобразование типов

Явные преобразования (приведения).

Для явных преобразований требуется выражение приведения.

Приведение требуется, если в ходе преобразования данные могут быть утрачены или преобразование может завершиться сбоем по другим причинам.

Типичными примерами являются числовое преобразование в тип с меньшей точностью или меньшим диапазоном и преобразование экземпляра базового класса в производный класс.

Преобразование типов

Пользовательские преобразования.

Такие преобразования выполняются специальными методами, которые можно определить для включения явных и неявных преобразований между пользовательскими типами без связи "базовый класс — производный класс"

Преобразование типов

Преобразования с использованием вспомогательных классов.

Используется, чтобы выполнить преобразование между несовместимыми типами, например целыми числами и объектами `System.DateTime`

Неявное преобразование типов

```
class Car  
{ }
```

```
class Bus:Car  
{ }
```

```
Car car = new Bus();
```

Явное преобразование типов

```
class Car  
{ }
```

```
class Bus:Car  
{ }
```

```
Bus bus;  
Car car = new Bus();  
bus = (Bus)car;
```


Ошибки преобразования типов

```
class Car  
{  
}
```

```
class Bus:Car  
{  
}
```

```
class Truck:Car  
{  
}
```

Ошибки преобразования типов

```
7  
8 Bus bus;  
9 Car car = new Truck();  
10 bus = (Bus)car;   
11  
12  
13
```

Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type 'ConsoleApp13.Truck' to type 'ConsoleApp13.Bus'.'

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)

▸ [Exception Settings](#)

Ошибки преобразования типов

```
Bus bus;  
Car car = new Truck();  
try  
{  
    bus = (Bus)car;  
}  
catch(InvalidCastException)  
{ }
```

Ключевое слово `as`

С помощью ключевого слова `as` программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение `null`:

Ключевое слово as

```
Person person1 = new Client("Bob", "RFB");  
var person2 = person1 as Client;
```

```
if (person2 != null)  
    { }  
else { }
```

Ключевое слово is

Ключевое слово `is` проверяет возможность преобразования одного типа в другой:

```
Person person1 = new Client("Bob", "RFB");  
Client person2;
```

```
if (person1 is Client)  
    { person2 = (Client)person1; }  
else { }
```