

ЛК 10 - ЛК 11. Полиморфизм

Полиморфизм

Полиморфизм

Класс-наследник может дополнять базовый класс новыми методами, а также замещать методы базового класса.

```
class Pet ...
{ public void Speak() ... }
class Dog : Pet...
{ public void Speak() ... }

. . .
Pet pet = new Pet();
Dog dog = new Dog();
pet.Speak();
dog.Speak();
```

Полиморфизм

Чтобы не было предупреждения, лучше

```
class Pet
{... public void Speak() ...}
class Dog : Pet
{... new public void Speak() ...}
...
Pet pet = new Pet();
Dog dog = new Dog();
pet.Speak();
dog.Speak();
```

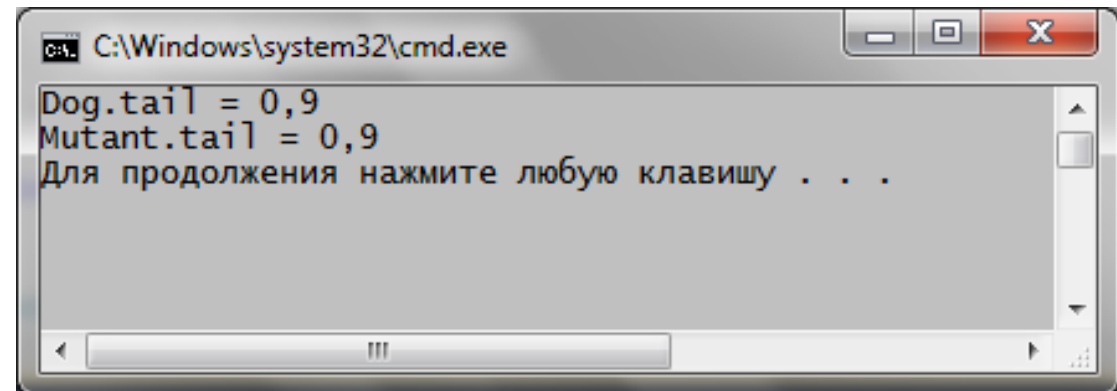
Полиморфизм

Ключевое слово `new` применимо и к полям класса.

```
class Dog : Pet
{... new public double tail = 0.9;  // хвост
    ... }
```

Теперь и для собаки и для мутанта:

```
Console.WriteLine("Dog.tail = "+ Dog.tail);
Console.WriteLine("Mutant.tail = "+ Mutant.tail);
```



Полиморфизм

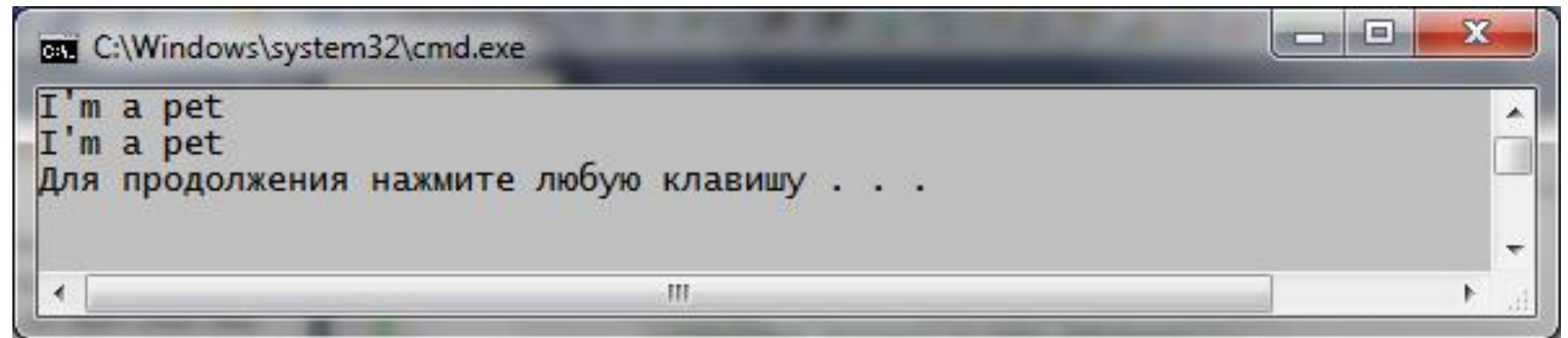
Такой способ называется **скрытием** или **замещением** члена класса.

Делая это явно, вы устанавливаете, что реализация производного типа преднамеренно спроектирована так, чтобы игнорировать родительскую версию (в реальном проекте это может помочь, если внешнее программное обеспечение .NET каким-то образом конфликтует с вашим программным обеспечением).

Полиморфизм

Замещение методов класса не является полиморфным по умолчанию.

```
Pet pet1, dog1;  
pet1 = new Pet();  
dog1 = new Dog(); // Допустимо по правилам присваивания  
pet1.Speak();  
dog1.Speak();
```



Полиморфизм

Использование слова `new` гарантирует только создание полей. Методы остаются привязанными к типу (раз класс объекта `Pet`, то и методы его).

Если создать массив (зверинец) – у нас будет массив типа `Pet`. В этом случае все животные в зверинце будут говорить одинаково.

Полиморфный интерфейс

Базовый класс конструируется таким образом, чтобы предоставить некоторый функционал наследникам с использованием любого количества виртуальных или абстрактных членов.

Полиморфный интерфейс

Виртуальный член — это член базового класса, определяющий реализацию по умолчанию, которая ***может быть изменена (переопределена)*** в производном классе.

Полиморфный интерфейс

Абстрактный метод — это член базового класса, который не предусматривает реализации по умолчанию, а предлагает только сигнатуру.

Когда класс наследуется от базового класса, определяющего абстрактный метод, этот метод **обязательно** должен быть переопределен в производном классе.

Полиморфный интерфейс

Для организации полиморфного вызова методов применяется пара ключевых слов `virtual` и `override`:

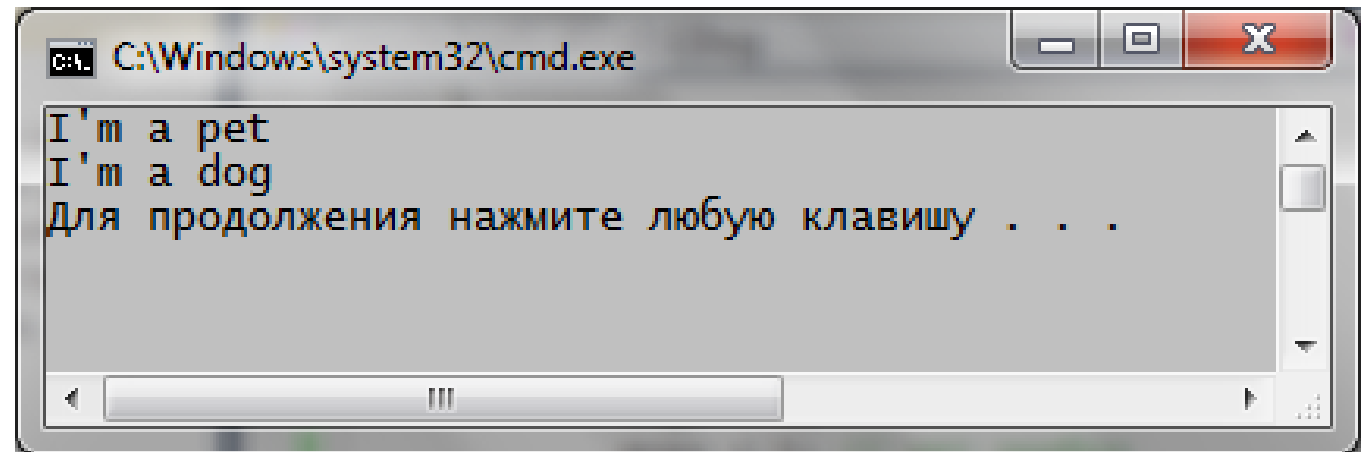
- ❑ `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным,
- ❑ `override` – для методов производных классов. Эти методы должны совпадать по имени и сигнатуре с перекрываемым методом класса-предка.

Полиморфный интерфейс

```
class Pet
{
    public virtual void Speak()
    {
        Console.WriteLine("I'm a pet");
    }
}
class Dog : Pet
{
    public override void Speak()
    {
        Console.WriteLine("I'm a dog");
    }
}
```

Полиморфный интерфейс

```
Pet pet1, dog1;  
pet1 = new Pet();  
dog1 = new Dog();  
pet1.Speak();  
dog1.Speak();
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window contains the following text:

```
I'm a pet  
I'm a dog  
Для продолжения нажмите любую клавишу . . .
```

Полиморфизм

Если на некоторой стадии построения иерархии классов требуется запретить дальнейшее переопределение виртуального метода в производных классах, этот метод помечается ключевым словом **sealed** (запечатанный):

```
class Dog : Pet
{ public sealed override void Speak() { ... }
}
```

И мутант не сможет сделать

```
public override void Speak()
```

(Но он сможет это сделать по-старому – без **override**).

Полиморфизм

Для методов абстрактных классов (классов с модификатором **abstract**) возможно задать модификатор **abstract**, который говорит о том, что метод не реализуется в классе, а должен обязательно переопределяться в наследнике.

```
abstract class AbstractClass
{
    //Реализации метода в классе нет
    public abstract void AbstractMethod();
}
```




Преобразование типов

Преобразование типов

- Создадим базовый класс и двух наследников

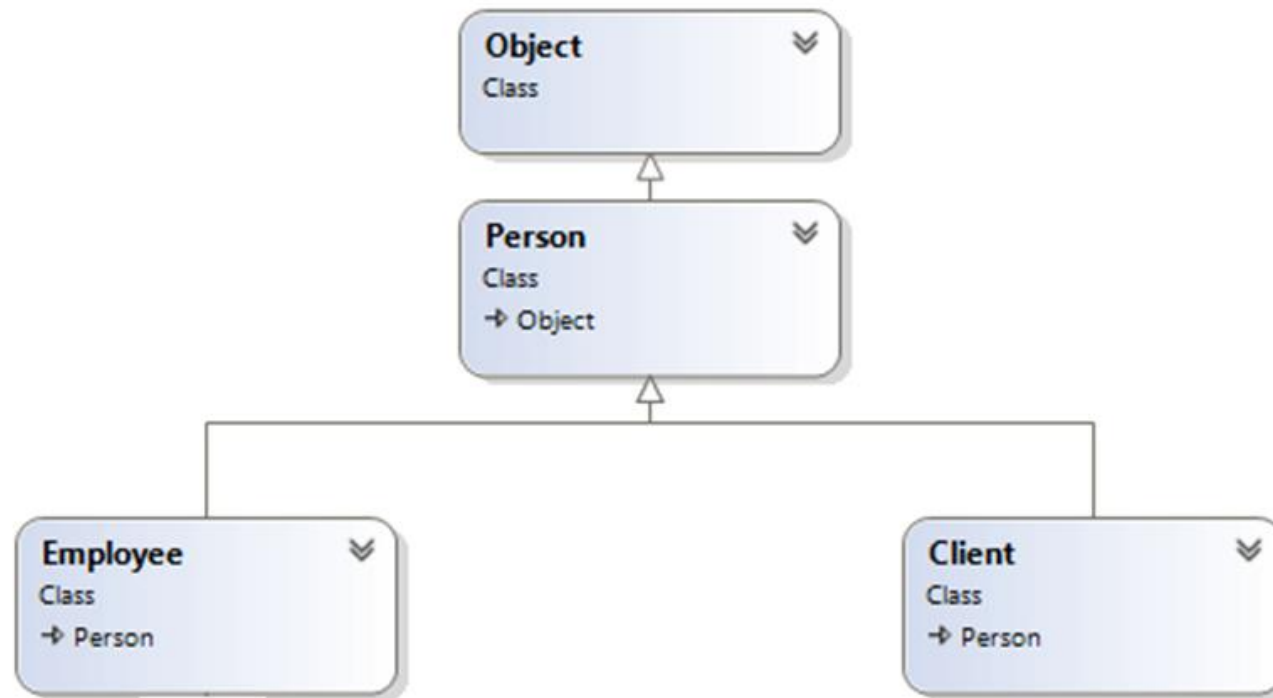
```
class Person
{
    ссылка: 2
    public string Name { get; set; }
    ссылка: 2
    public Person(string name)
    {
        Name = name;
    }
    ссылка: 0
    public void Display()
    {
        Console.WriteLine($"Person {Name}");
    }
}
```

```
class Employee : Person
{
    ссылка: 1
    public string Company { get; set; }
    ссылка: 0
    public Employee(string name, string company) : base(name)
    {
        Company = company;
    }
}
```

```
class Client : Person
{
    ссылка: 1
    public string Bank { get; set; }
    ссылка: 0
    public Client(string name, string bank) : base(name)
    {
        Bank = bank;
    }
}
```

Преобразование типов

- В этой иерархии классов мы можем проследить следующую цепь наследования: **Object** (все классы неявно наследуются от типа **Object**) -> **Person** -> **Employee|Client**.



Восходящие преобразования. Upcasting

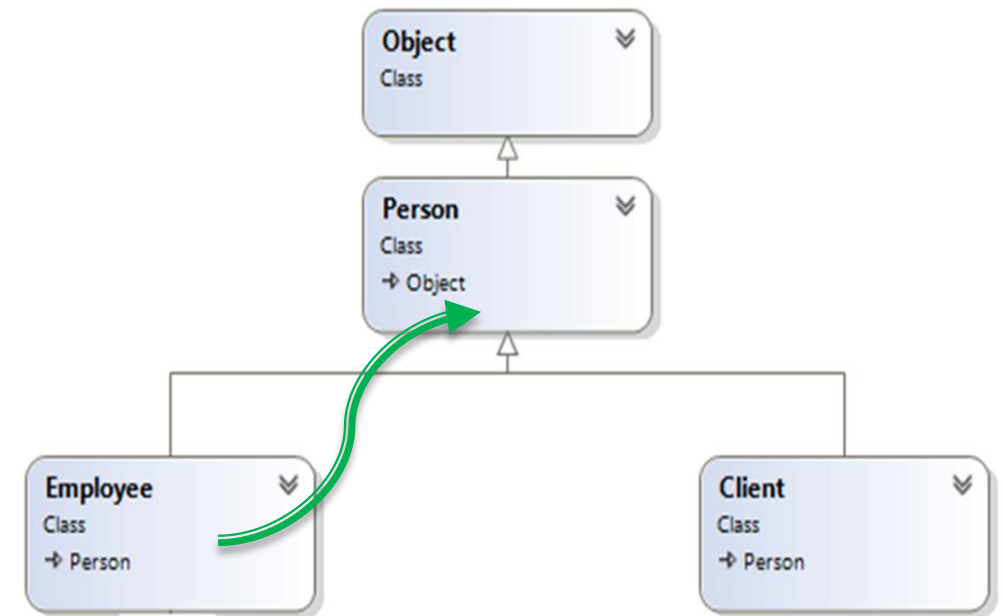
- Объекты производного типа (который находится внизу иерархии) в то же время представляют и базовый тип.
- Например, объект `Employee` в то же время является и объектом класса `Person`.
- Что в принципе естественно, так как каждый сотрудник (`Employee`) является человеком (`Person`).

```
static void Main(string[] args)
{
    Employee employee = new Employee("Tom", "Microsoft");
    Person person = employee;    // преобразование от Employee к Person

    Console.WriteLine(person.Name);
    Console.ReadKey();
}
```

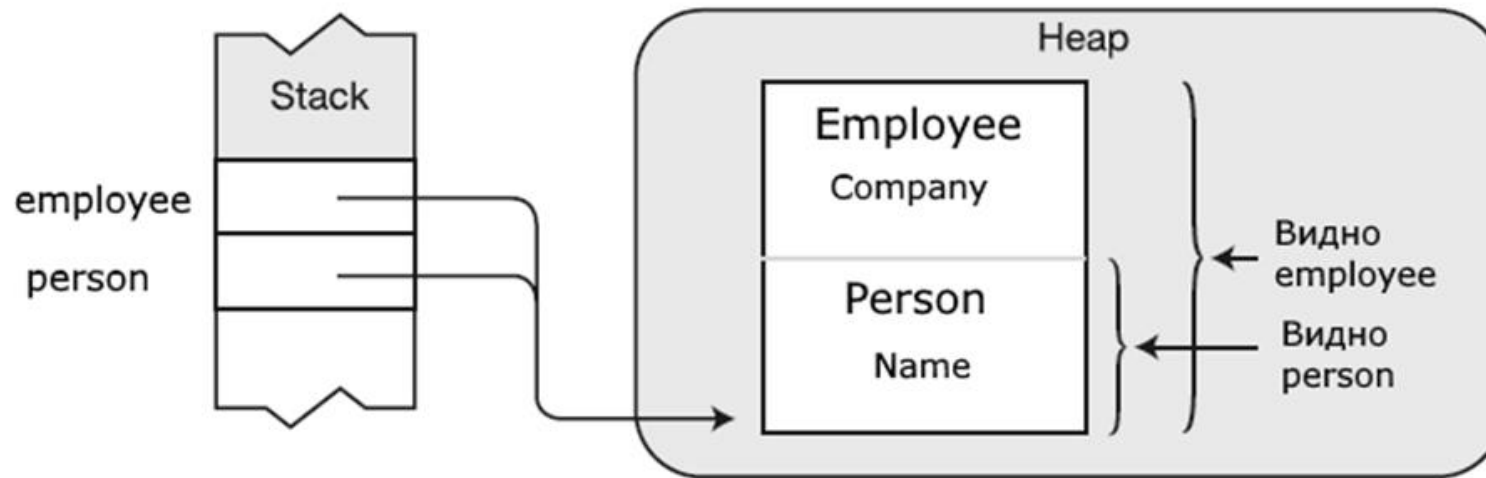
Восходящие преобразования. Upcasting

- В данном случае переменной `person`, которая представляет тип `Person`, присваивается ссылка на объект `Employee`.
- Но чтобы сохранить ссылку на объект одного класса в переменную другого класса, необходимо выполнить преобразование типов - в данном случае от типа `Employee` к типу `Person`.
- И так как `Employee` наследуется от класса `Person`, то автоматически выполняется неявное восходящее преобразование - преобразование к типу, которые находятся выше иерархии классов, то есть к базовому классу.



Восходящие преобразования. Upcasting

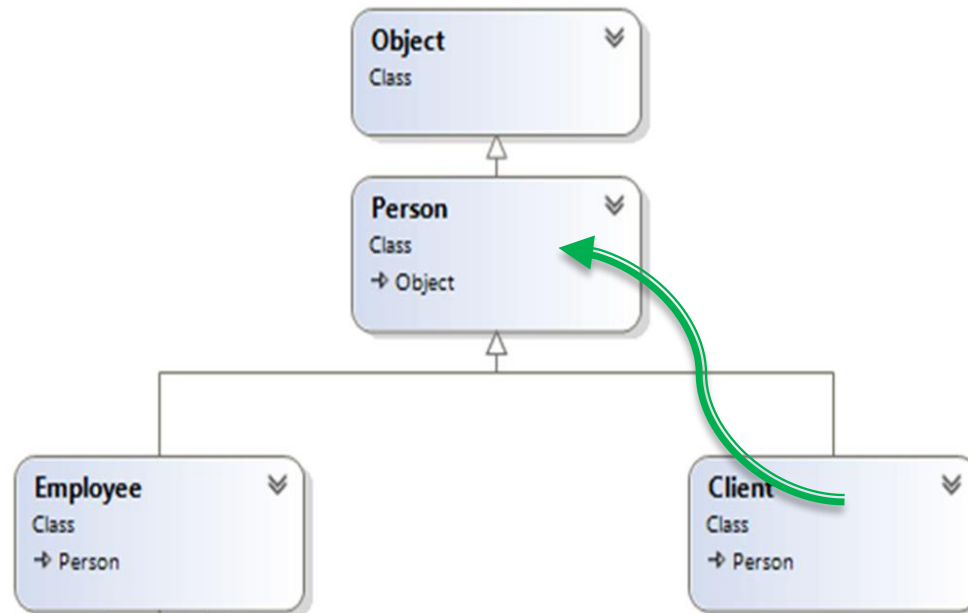
- В итоге переменные `employee` и `person` будут указывать на один и тот же объект в памяти, но переменной `person` будет доступна только та часть, которая представляет функционал типа `Person`.



Восходящие преобразования. Upcasting

- Здесь переменная `person2`, которая представляет тип `Person`, хранит ссылку на объект `Client`, поэтому также выполняется восходящее неявное преобразование от производного класса `Client` к базовому типу `Person`.

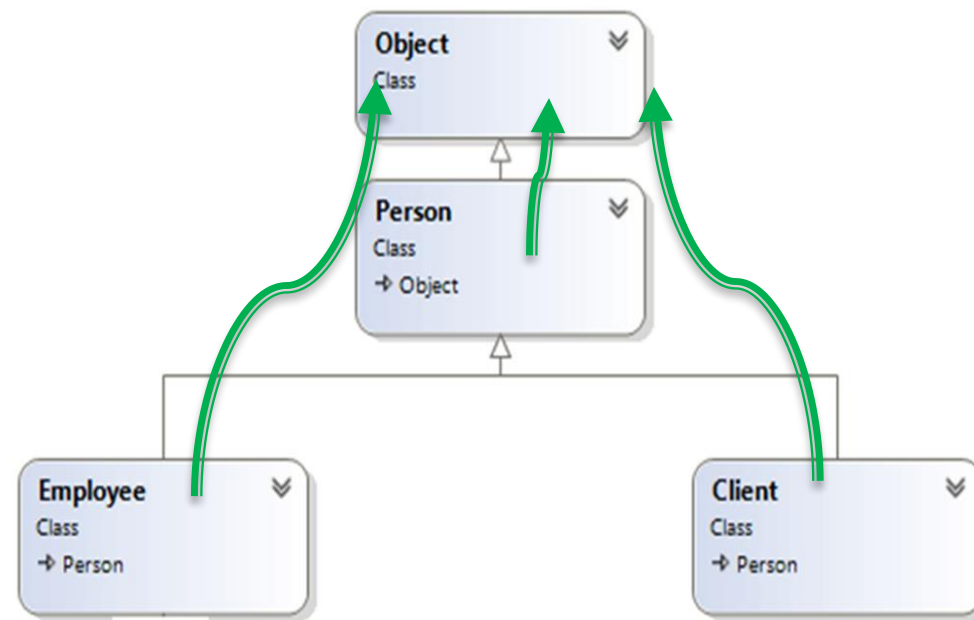
```
Person person2 = new Client("Bob", "ContosoBank"); // преобразование от Client к Person
```



Восходящие преобразования. Upcasting

- Так как тип `object` - базовый для всех остальных типов, то преобразование к нему будет производиться автоматически.

```
object person1 = new Employee("Tom", "Microsoft"); // от Employee к object
object person2 = new Client("Bob", "ContosoBank"); // от Client к object
object person3 = new Person("Sam");                // от Person к object
```



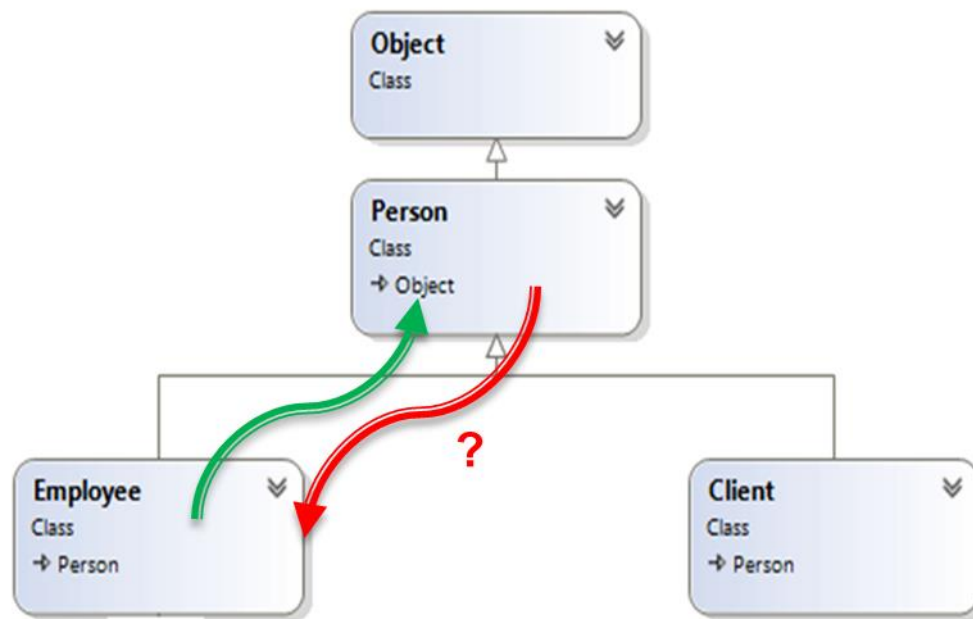
Нисходящие преобразования. Downcasting

- Но кроме восходящих преобразований от производного к базовому типу есть нисходящие преобразования или downcasting - от базового типа к производному.

```
Employee employee = new Employee("Tom", "Microsoft");  
Person person = employee;    // преобразование от Employee к Person
```

Нисходящие преобразования. Downcasting

- И может возникнуть вопрос, можно ли обратиться к функционалу типа **Employee** через переменную типа **Person**.
- Но автоматически такие преобразования не проходят, ведь не каждый человек (объект **Person**) является сотрудником предприятия (объектом **Employee**).



Нисходящие преобразования. Downcasting

- И для нисходящего преобразования необходимо применить явное преобразования, указав в скобках тип, к которому нужно выполнить преобразование:

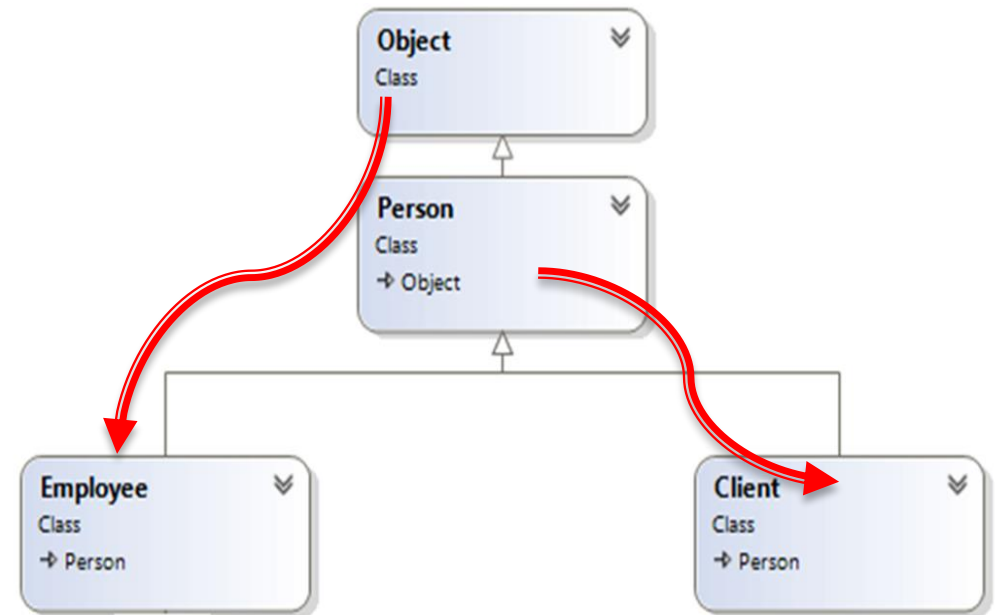
```
Employee employee = new Employee("Tom", "Microsoft");  
Person person = employee;    // преобразование от Employee к Person  
  
//Employee employee2 = person;    // так нельзя, нужно явное преобразование  
Employee employee2 = (Employee)person;    // преобразование от Person к Employee
```

Примеры преобразований:

```
// Объект Employee также представляет тип object  
object obj = new Employee("Bill", "Microsoft");
```

```
// чтобы обратиться к возможностям типа Employee, приводим объект к типу Employee  
Employee emp = (Employee)obj;
```

```
// объект Client также представляет тип Person  
Person person = new Client("Sam", "ContosoBank");  
// преобразование от типа Person к Client  
Client client = (Client)person;
```



Примеры преобразований:

- Если нам надо обратиться к каким-то отдельным свойствам или методам объекта, то нам необязательно присваивать преобразованный объект переменной :

```
// Объект Employee также представляет тип object  
object obj = new Employee("Bill", "Microsoft");
```

```
// преобразование к типу Person для вызова метода Display  
((Person)obj).Display();  
// либо так  
// ((Employee)obj).Display();
```

```
// преобразование к типу Employee, чтобы получить свойство Company  
string comp = ((Employee)obj).Company;
```

Примеры преобразований:

- В то же время необходимо соблюдать осторожность при подобных преобразованиях.

```
// Объект Employee также представляет тип object  
object obj = new Employee("Bill", "Microsoft");  
  
// преобразование к типу Client, чтобы получить свойство Bank  
string bank = ((Client)obj).Bank;
```

Исключение не обработано

System.InvalidCastException: "Не удалось привести тип объекта "CCar.Employee" к типу "CCar.Client"."

[Просмотреть сведения](#) | [Копировать подробности](#)

▸ [Параметры исключений](#)

```
Console.ReadKey();
```

Примеры преобразований:

```
Employee emp = new Person("Tom"); // ! Ошибка
```

```
Person person = new Pers  
Employee emp2 = (Employee
```

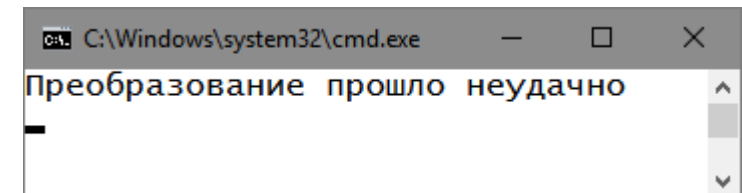
Person.Person(string name)

Не удастся неявно преобразовать тип "CCar.Person" в "CCar.Employee". Существует явное преобразование (возможно, пропущено приведение типов).

Способы преобразований

- Во-первых, можно использовать ключевое слово `as`. С помощью него программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение `null`:

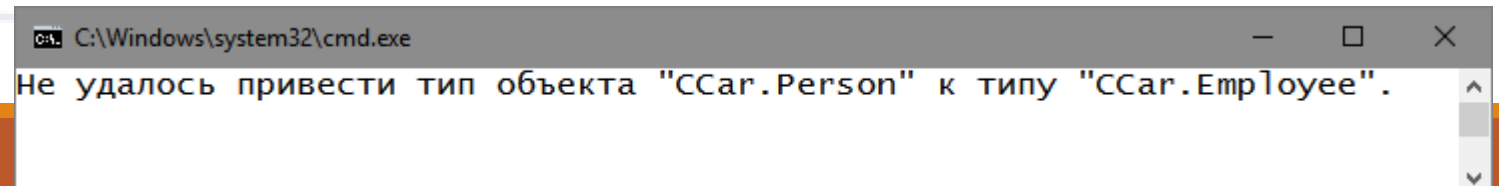
```
Person person = new Person("Tom");
Employee emp = person as Employee;
if (emp == null)
{
    Console.WriteLine("Преобразование прошло неудачно");
}
else
{
    Console.WriteLine(emp.Company);
}
```



Способы преобразований

- Второй способ заключается в отлавливании исключения `InvalidCastException`, которое возникнет в результате преобразования:

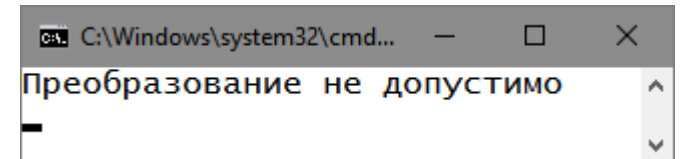
```
Person person = new Person("Tom");  
try  
{  
    Employee emp = (Employee)person;  
    Console.WriteLine(emp.Company);  
}  
catch (InvalidCastException ex)  
{  
    Console.WriteLine(ex.Message);  
}
```



Способы преобразований

- Третий способ заключается в проверке допустимости преобразования с помощью ключевого слова **is**:

```
Person person = new Person("Tom");
if (person is Employee)
{
    Employee emp = (Employee)person;
    Console.WriteLine(emp.Company);
}
else
{
    Console.WriteLine("Преобразование не допустимо");
}
```





Различие переопределения и сокрытия методов

Различие переопределения и сокрытия методов

- Ранее было рассмотрена два способа изменения функциональности методов, унаследованных от базового класса - сокрытие и переопределение.
- В чем разница между двумя этими способами?

Различие переопределения и сокрытия методов

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

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

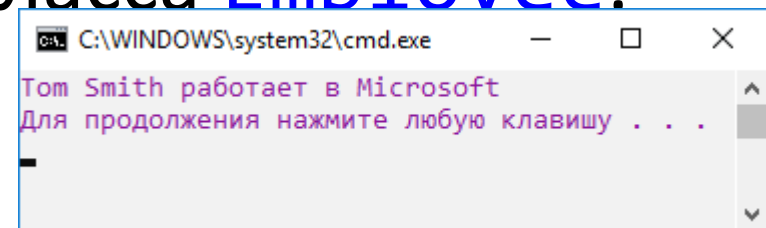
- Снова рассмотрим классы **Person** и **Employee**. Метод **Display()** переопределен (он виртуальный)

Различие переопределения и сокрытия методов

- Также создадим объект **Employee** и передадим его переменной типа **Person**:

```
Person tom = new Employee("Tom", "Smith", "Microsoft");  
tom.Display();           // Tom Smith работает в Microsoft
```

- Теперь мы получаем иной результат, нежели при сокрытии. А при вызове **tom.Display()** выполняется реализация метода **Display** из класса **Employee**.



Различие переопределения и сокрытия методов

- Для работы с виртуальными методами компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записываются адреса виртуальных методов. Для каждого класса создается своя таблица.
- Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT.

Различие переопределения и сокрытия методов

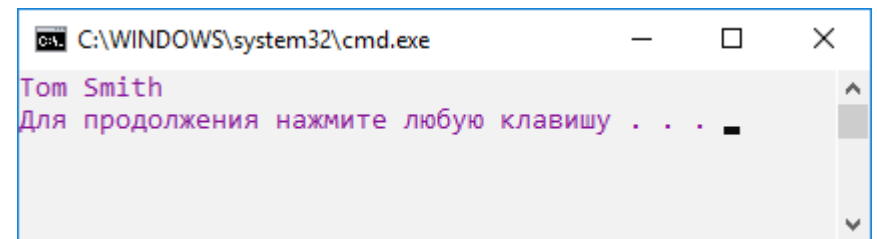
- А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление.
- То есть процесс выбора реализации метода производится во время выполнения программы.
- Собственно так и выполняется виртуальный метод.
- Следует учитывать, что так как среде выполнения вначале необходимо получить из таблицы VMT адрес нужного метода, то это немного замедляет выполнение программы.

Различие переопределения и сокрытия методов

- **Соккрытие**

- Теперь возьмем те же классы **Person** и **Employee**, но вместо переопределения используем сокрытие:

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



Различие переопределения и сокрытия методов

- Переменная `tom` представляет тип `Person`, но хранит ссылку на объект `Employee`.
- Однако при вызове метода `Display` будет выполняться та версия метода, которая определена именно в классе `Person`, а не в классе `Employee`.
- Почему?
- Класс `Employee` никак не переопределяет метод `Display`, унаследованный от базового класса, а фактически определяет новый метод.
- Поэтому при вызове `tom.Display()` вызывается метод `Display` из класса `Person`.



Абстрактные классы

Абстрактные классы.

Создадим для наших животных новый метод – кормление. Мы будем их кормить, а они нам скажут, как им понравилось. Сделаем метод кормления абстрактным. Тогда и класс должен быть абстрактным (иначе что же получится, если мы создадим объект такого класса?).

```
abstract class Pet
{ ...
    public abstract string Eat(float meat, float grain, float
    vegetable);
}
```

Абстрактные классы.

Методы, помеченные как `abstract`, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости).

Здесь абстрактный класс `Pet` информирует типы-наследники о том, что у него есть метод по имени `Eat()`, с такими-то аргументами, который возвращает строку.

О необходимых деталях должен позаботиться наследник.

Абстрактные классы.

ConsoleApplication1 - Microsoft Visual Studio (Administrator)

File Edit View Refactor Project Build Debug Team Data Tools Architecture Test Analyze Window Help

Toolbox

General

There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

Program.cs

ConsoleApplication1.Program

Main(string[] args)

```
    }  
    new public void Speak()  
    {  
        base.Speak();  
        Console.WriteLine("I'm a mutant");  
    }  
}  
class CFish : CPet  
{  
    public int fins; // плавник  
    public CFish(int eyes, int tail, int fins)  
    {  
    }  
}
```

100 %

Error List

3 Errors 0 Warnings 0 Messages

	Description	File	Line	Column	Project
1	'ConsoleApplication1.CDog' does not implement inherited abstract member 'ConsoleApplication1.CPet.Eat(float, float, float)'	Program.cs	25	11	ConsoleApplication1
2	'ConsoleApplication1.CBird' does not implement inherited abstract member 'ConsoleApplication1.CPet.Eat(float, float, float)'	Program.cs	40	11	ConsoleApplication1

Ready

Ln 87 Col 31 Ch 31 INS

Абстрактные классы.

Как минимум, нужно описать этот метод у прямых наследников:

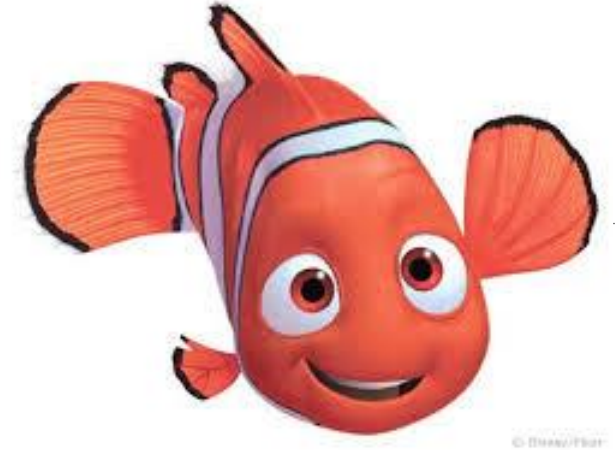
```
class Dog : Pet
{
    ...
    public override string Eat(float meat, float grain, float
vegetable)
    {
        if (grain > 0) return "I don't eat grain";
        if (meat < 1 & vegetable < 1) return "There is nothing ";
        return "I'm pleased";
    }
}
```

Абстрактные классы.



```
class Bird : Pet
{
    ...
    public override string Eat(float meat, float grain, float
vegetable)
    {
        if (meat > 0) return "I don't eat meat";
        if (grain < 1 & vegetable < 1) return "There is nothing ";
        return "I'm pleased";
    }
}
```

Абстрактные классы.

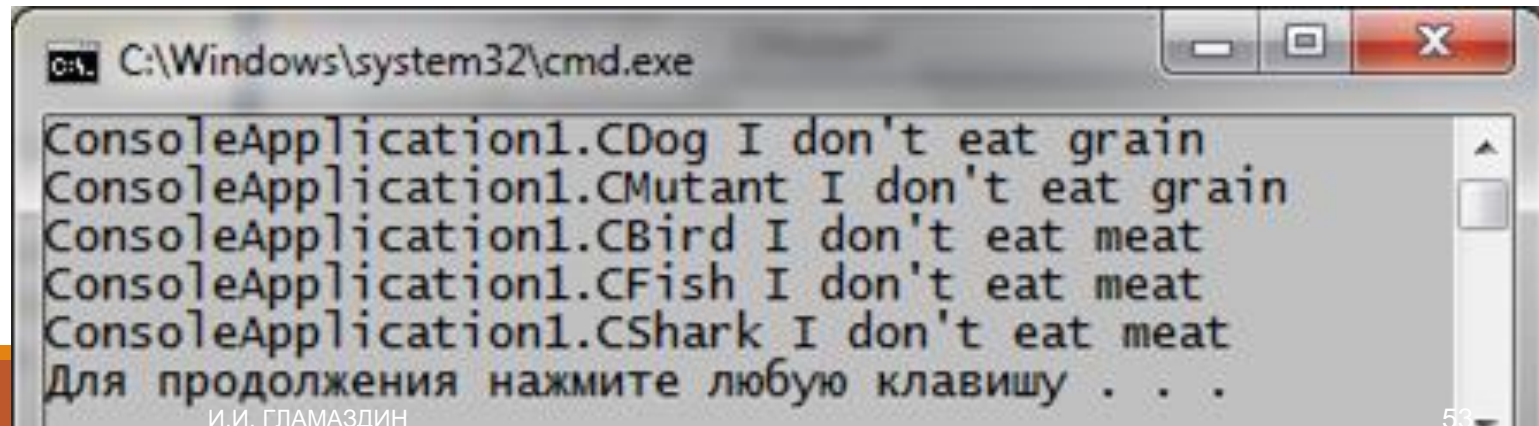


```
class Fish : Pet
{
    ...
    public override string Eat(float meat, float grain, float
vegetable)
{
    if (meat > 0) return "I don't eat meat";
        if (grain < 1 & vegetable < 1) return "There is nothing ";
        return "I'm pleased";
}
```

Абстрактные классы.

А теперь в функции Main создадим зоопарк (обратите внимание, что рыбка и акула требуют особых конструкторов):

```
Pet[] Zoo = { new Dog(), new Mutant(), new Bird(),  
              new Fish(2,1,4), new Shark(2,1,4,1000) };  
foreach (Pet s in Zoo)  
{  
    Console.WriteLine(s.GetType() + " ");  
    Console.WriteLine(s.Eat(100,100,100));  
}
```



```
C:\Windows\system32\cmd.exe  
ConsoleApplication1.CDog I don't eat grain  
ConsoleApplication1.CMutant I don't eat grain  
ConsoleApplication1.CBird I don't eat meat  
ConsoleApplication1.CFish I don't eat meat  
ConsoleApplication1.CShark I don't eat meat  
Для продолжения нажмите любую клавишу . . .
```

Полиморфизм

Можно также описать этот метод и для более далеких наследников:

```
class CMutant : CDog
{
    ...
    public override string Eat(float meat, float grain, float vegetable)
    {
        if (meat < 100 & grain < 100 & vegetable < 100)
            return "There is nothing";
        return "that will do";
    }
}
```



Полиморфизм

```
class CShark : CFish
```

```
{
```

```
...
```

```
public override string Eat(float meat, float grain, float vegetable)
```

```
{
```

```
    if (meat < 1000) return "There is nothing";
```

```
    return "I'm pleased";
```

```
}
```



```
C:\Windows\system32\cmd.exe

ConsoleApplication1.CDog I don't eat grain
ConsoleApplication1.CMutant that will do
ConsoleApplication1.CBird I don't eat meat
ConsoleApplication1.CFish I don't eat meat
ConsoleApplication1.CShark There is nothing
Для продолжения нажмите любую клавишу . . .
```

Полиморфизм

Класс может не реализовывать абстрактный метод только в том случае, если он сам считается абстрактным. И тогда реализация достанется его наследникам.

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



Полиморфизм

Мы просто объявили рыбку абстрактной и удалили из нее реализацию метода `Eat`.

Но теперь не может существовать просто рыбка – только акула. Удалим рыбку из зоопарка.

```
Pet[] Zoo = {new Dog(), new Mutant(),  
             new Bird(), new Shark(2,1,4,1000) };
```

Абстрактные члены классов

- Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова **abstract** и не имеют никакого функционала. В частности, абстрактными могут быть:
 - Методы
 - Свойства
 - Индексаторы
 - События

Абстрактные свойства

```
abstract class Person
{
    2 references
    public abstract string Name { get; set; }
}
```

0 references

```
class Client : Person
```

```
{
    private string name;

    2 references
    public override string Name
    {
        get { return "Mr/Ms. " + name; }
        set { name = value; }
    }
}
```

1 reference

```
class Employee : Person
```

```
{
    2 references
    public override string Name { get; set; }
}
```

Отказ от реализации абстрактных членов

```
abstract class Person
{
    0 references
    public abstract string Name { get; set; }
}
```

```
1 reference
abstract class Manager : Person
{
}
```



Тест

Почему следующая программа не компилируется?

```
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Employee();
        Console.ReadKey();
    }
}
internal class Person
{
}
public class Employee : Person
{
}
```

Почему следующая программа не компилируется?

```
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Employee();
        Console.ReadKey();
    }
}
internal class Person
{
}
public class Employee : Person
{
}
```

- Наследник не может расширять доступ

Следующая программа не компилируется.
В чем ошибка?

```
class Base
{
    public virtual void Display()
    {
        Console.WriteLine("This is Base");
    }
}
abstract class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine("This is Derived");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Base _base = new Derived();
        _base.Display();
        Console.ReadKey();
    }
}
```

Следующая программа не компилируется.
В чем ошибка?

```
class Base
{
    public virtual void Display()
    {
        Console.WriteLine("This is Base");
    }
}
abstract class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine("This is Derived");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Base _base = new Derived();
        _base.Display();
        Console.ReadKey();
    }
}
```

Нельзя создать объект
абстрактного класса

Есть ли в следующем коде ошибка? Если есть, то какая?

```
abstract class Base
{
    public abstract void Display();
}
abstract class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine("This is Derived");
    }
}
```

Есть ли в следующем коде ошибка? Если есть, то какая?

```
abstract class Base
{
    public abstract void Display();
}
abstract class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine("This is Derived");
    }
}
```

Все верно

Что будет выведено на консоль в результате выполнения следующей программы и почему?

```
abstract class Base
{
    protected string name = "Base";
    public abstract void Display();
    protected abstract void Do();
}
class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine(name);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Base _base = new Derived();
        _base.Display();
        Console.ReadKey();
    }
}
```

Что будет выведено на консоль в результате выполнения следующей программы и почему?

```
abstract class Base
{
    protected string name = "Base";
    public abstract void Display();
    protected abstract void Do();
}
class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine(name);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Base _base = new Derived();
        _base.Display();
        Console.ReadKey();
    }
}
```

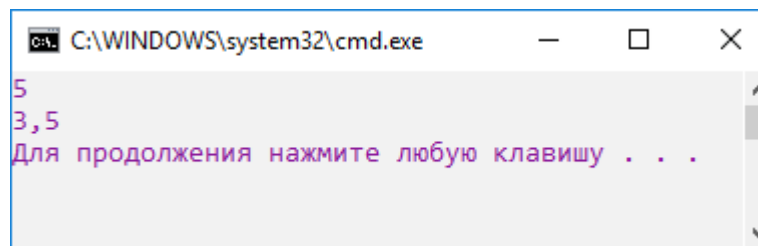
Ошибка!
не все абстрактные методы
реализованы

Класс System.Object и его методы

- Метод **ToString** служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

```
int i = 5;  
Console.WriteLine(i.ToString()); // выведет число 5
```

```
double d = 3.5;  
Console.WriteLine(d.ToString()); // выведет число 3,5
```



Класс System.Object и его методы

- Метод **GetHashCode** позволяет вернуть некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализации базового типа:

```
class Person
{
    1 reference
    public string Name { get; set; }

    1 reference
    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
}
```


Класс System.Object и его методы

- Метод **GetType** позволяет получить тип данного объекта

```
Person person = new Person { Name = "Tom" };  
Console.WriteLine(person.GetType());    // Person
```

Класс System.Object и его методы

- Метод **Equals** позволяет сравнить два объекта на равенство:

```
class Person
{
    3 references
    public string Name { get; set; }
    0 references
    public override bool Equals(object obj)
    {
        if (obj.GetType() != this.GetType()) return false;

        Person person = (Person)obj;
        return (this.Name == person.Name);
    }
}
```

```
Person person1 = new Person { Name = "Tom" };
Person person2 = new Person { Name = "Bob" };
Person person3 = new Person { Name = "Tom" };
bool p1Ep2 = person1.Equals(person2);    // false
bool p1Ep3 = person1.Equals(person3);    // true
```

