ООП

otus.ru

Проверить, идет ли запись

Напишите «+» в чат, если меня слышно и видно

Тема урока

ООП

Нилов Павел

Преподаватель курса C# Professional, C# Basic

Контакты:t.me/@NilovPavel

Правила вебинара

Активно
участвуем

Задаем вопрос
в чат

Вопросы вижу в чате,
могу ответить не сразу

Цели вебинара

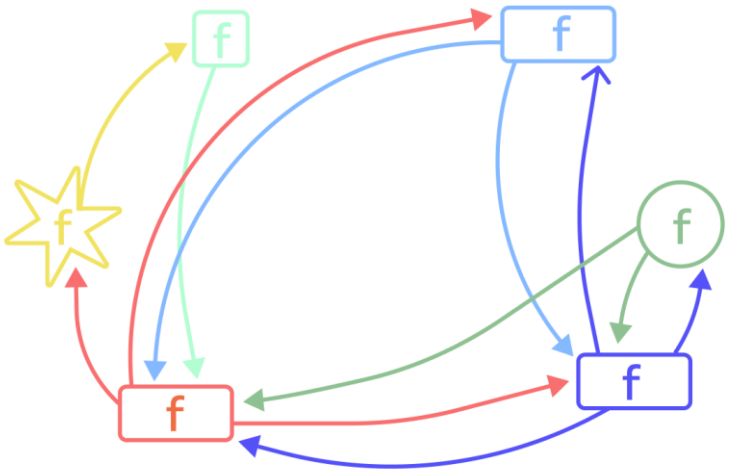
1. Узнать, что такое ООП.
2. Основные механизмы ООП.
3. Особенности синтаксиса ООП.

Кратко об ООП

Функциональное программирование

До ООП в разработке использовался другой подход — процедурный. Программа представляется в нем как набор процедур и функций — подпрограмм, которые выполняют определенный блок кода с нужными входящими данными.

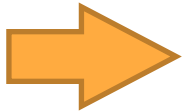
Процедурное программирование хорошо подходит для легких программ без сложной структуры. Но если блоки кода большие, а функций сотни, придется редактировать каждую из них, продумывать новую логику. В результате может образоваться много плохо читаемого, перемешанного кода — «спагетти-кода» или «лапши».



Функциональное программирование

Код ассемблера:

```
mov ax, 0x6h          ; заносим в AX число 6
mov cx, 0x8h          ; заносим в AX число 8
mov dx, cx            ; копируем CX в DX, DX = 6
add dx, ax            ; DX = DX + AX
```



Код C:

```
int main() {
    int a = 6;
    int b = 8;
    int sum = a + b;
    return 0;
}
```

Минусы использования функционального программирования **для крупных проектов**:

1. Сложно управлять кодом в виду того, что становится сложно декомпозировать код.
2. Отсутствовала инкапсуляция, в виду этого была нарушена связь между кодом и данными.
3. Необходимо было копировать код и, как следствие, его дублировать.

Что такое ООП?

Объектно-ориентированное программирование (ООП) — это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение.
ООП – это программирование с помощью классов и объектов.

Всё вокруг нас является **объектом.**

У объекта есть **свойства** (еще называют параметры)

У объекта есть **методы** (методы – это действия, то есть что может делать объект)

Например, машина – это объект.

У любой машины есть такие свойства: модель, цвет, размер и т.д.

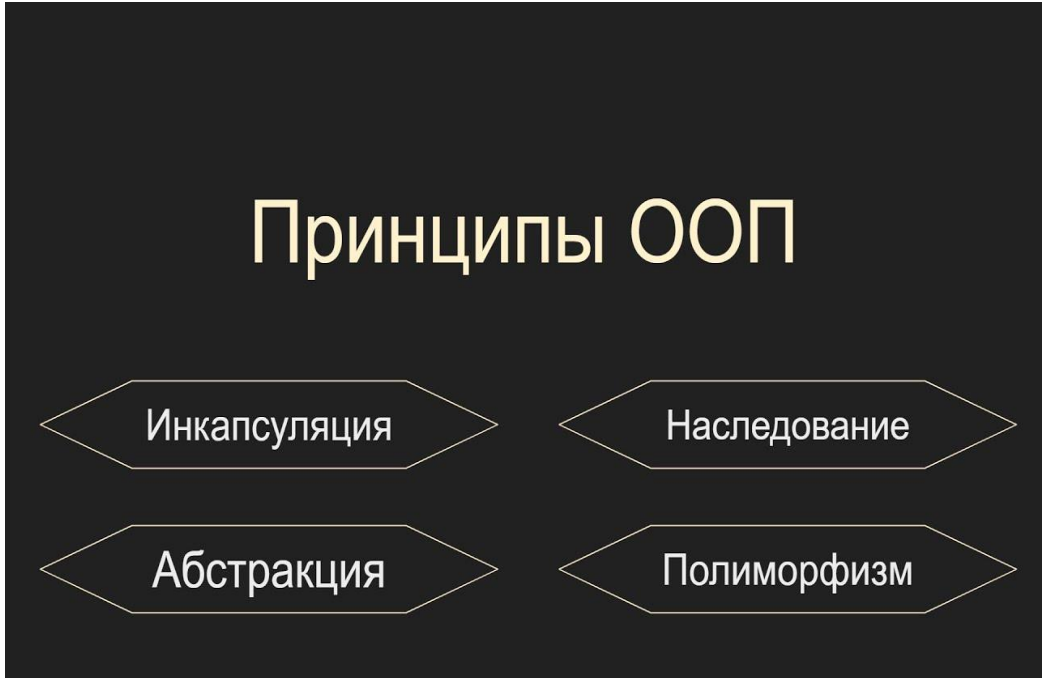
Методы машины:
затормозить ();
нажатьНаГаз ();
открытьДверь ();
закрытьДверь ();
и т.д.

Например, котёнок – это объект.

У любого котенка есть свойства: порода, имя, цвет, длина шерсти, возраст и т.д.

Методы котенка:
спать();
кушать();
играть ();
шкодить ();
и т.д.

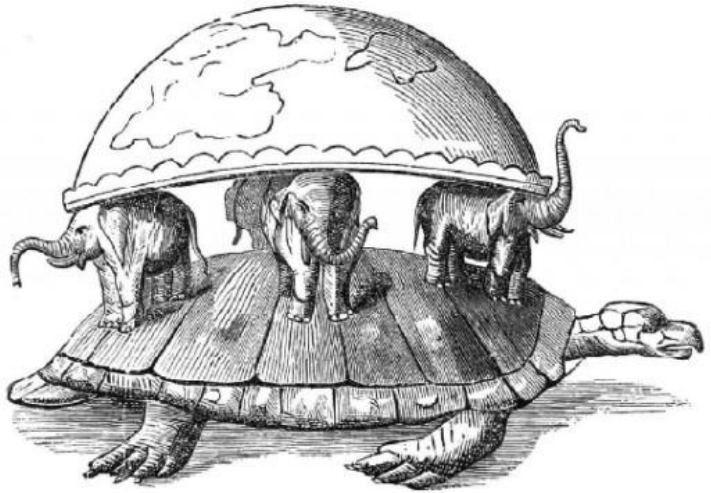
Что из перечисленного ниже НЕ является принципом ООП?



Принципы ООП

Объектно-ориентированное программирование основано на следующих принципах:

1. **Инкапсуляция**
2. **Наследование**
3. **Полиморфизм (типов)**
4. **Абстракция**

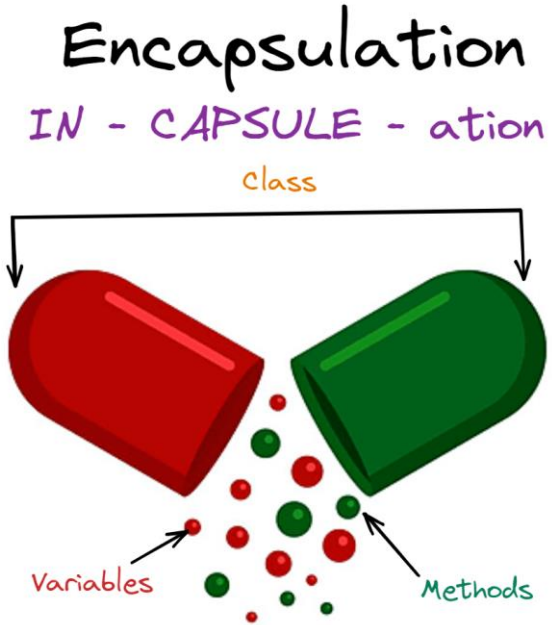


Инкапсуляция

Что такое инкапсуляция?

Инкапсуляция заключается в объединении данных и методов, которые с ними работают, внутри одного объекта.

При этом доступ к данным объекта ограничивается (через модификаторы доступа), чтобы скрыть внутреннюю реализацию и предоставить только необходимый интерфейс.



Содержимое класса

Класс может содержать:

1. Поля (Fields)
2. Свойства (Properties)
3. Методы (Methods)
4. Конструкторы (Constructors)

Класс:
программист

Объект:
разработчик Иван

Атрибуты:
зарплата, обязанности

Методы:
написание кода

```
class Coder
{
    private string name;
    private string salary;

    public string Name
    {
        get { return this.name;
}
    }

    public void WriteCode()
    {
        //something to do
    }

    public Coder(string name)
    {
        this.name = name;
    }
}
```

Права доступа

Права доступа к членам класса в программировании нужны для управления видимостью и доступом к данным и методам объекта. Они обеспечивают **инкапсуляцию**, один из ключевых принципов объектно-ориентированного программирования (ООП), помогая скрыть внутреннюю реализацию объекта и предоставляя только необходимый для работы интерфейс.

Модификатор	Описание
public	Член доступен из любого места, где виден объект.
private	Член доступен только внутри класса, в котором он определён. Это значение по умолчанию.
protected	Член доступен в классе и его наследниках.
internal	Член доступен внутри текущей сборки (assembly).

Модификаторы доступа на бытовых примерах

public предоставляет возможность миру взаимодействовать с созданным объектом.

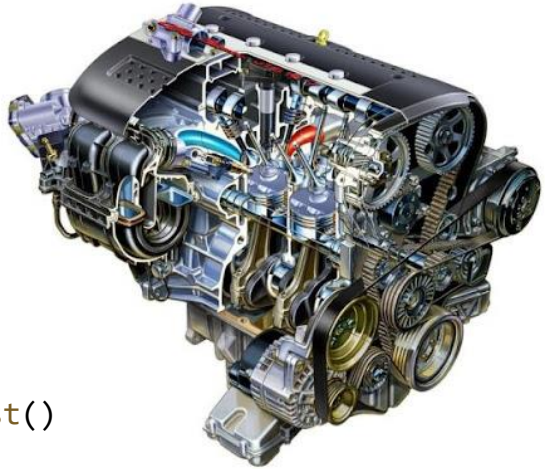
Пример: завести автомобиль ключом зажигания. *Для того, чтобы запустить двигатель, человек не должен залезать в двигатель, у него есть интерфейс (публичный метод запуска двигателя).*

```
class Car
{
    public void RunEngine()
    {  … }
}
```

Модификаторы доступа на бытовых примерах

private не предоставляет миру возможность взаимодействовать с созданным объектом, а служит только для внутреннего использования.

Пример: двигатель автомобиля передает крутящий момент(GetTwist()). Все, что происходит внутри двигателя: подача топлива, подача воздуха и пр. – скрыто от мира. Все узлы, агрегаты, характеристики – тоже.



```
class Engine
{
    public void GetTwist()
    {  … }

    private void GetAir()
    {  … }

    private int angularVelocity;
}
```

Модификаторы доступа на бытовых примерах

protected не предоставляет миру возможность взаимодействовать с созданным объектом и служит только для использования в наследовании.

Пример: все (в любом случае большинство) автомобили состоят из одних и тех же элементов. Скажем, что у любого автомобиля есть такой показатель, как скорость, но у разных автомобилей она может быть разной.


```
class Car
{
    protected virtual
            int GetSpeed()
    {
        return 0;
    }
}

class SuperCar : Car
{
    protected override
            int GetSpeed()
    {
        return 200;
    }
}
```

Инкапсуляция дает следующую информацию

Итого, инкапсуляция определяет:

1. Как выглядит создаваемый объект во внешнем мире.
2. Какими характеристиками он обладает.
3. Какое поведение он реализует.

Решение задач

Задачи:
1. Опишите класс Student, который:
1.1 Содержит имя.
1.2 Предоставляет имя, через свойство.
1.3 Содержит возраст.
1.4 Предоставляет возраст через метод.
1.5 Содержит конструктор, принимающий имя студента и возраст.
1.6 Содержит поле, показывающее, является ли студент совершеннолетним.
1.7 При инициализации поля имени в конструкторе вычисляет, является ли студент совершеннолетним в приватном методе.

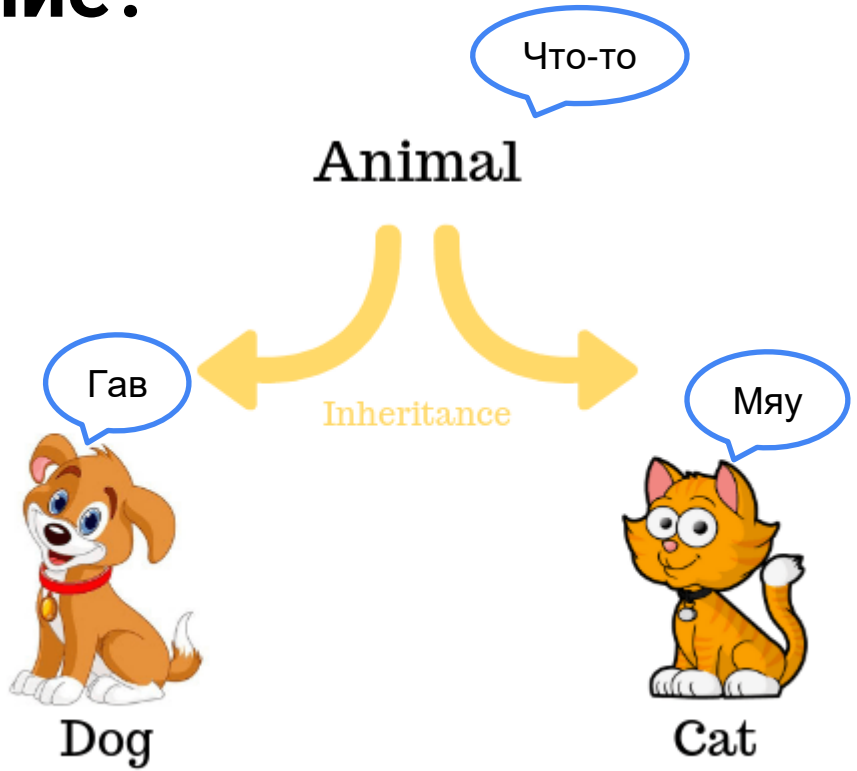
2. Опишите класс Dog
2.1 Содержит имя.
2.2 Содержит состояние: enum { sit, lie, stand }
2.3 Содержит метод, выполняющий команду "{Имя}, сидеть". Меняет состояние собаки на соответствующее.
2.4 Содержит метод, выполняющий команду "{Имя}, лежать". Меняет состояние собаки на соответствующее.

Наследование

Что такое наследование?

Наследование – это механизм, который позволяет использовать возможности других классов.

Наследование позволяет определить дочерний класс, который использует (наследует), расширяет или изменяет возможности родительского класса. Класс, члены которого наследуются, называется *базовым классом*. Класс, который наследует члены базового класса, называется *производным классом*.



Синтаксис наследования

Наследоваться в рамках языка C# допустимо от:

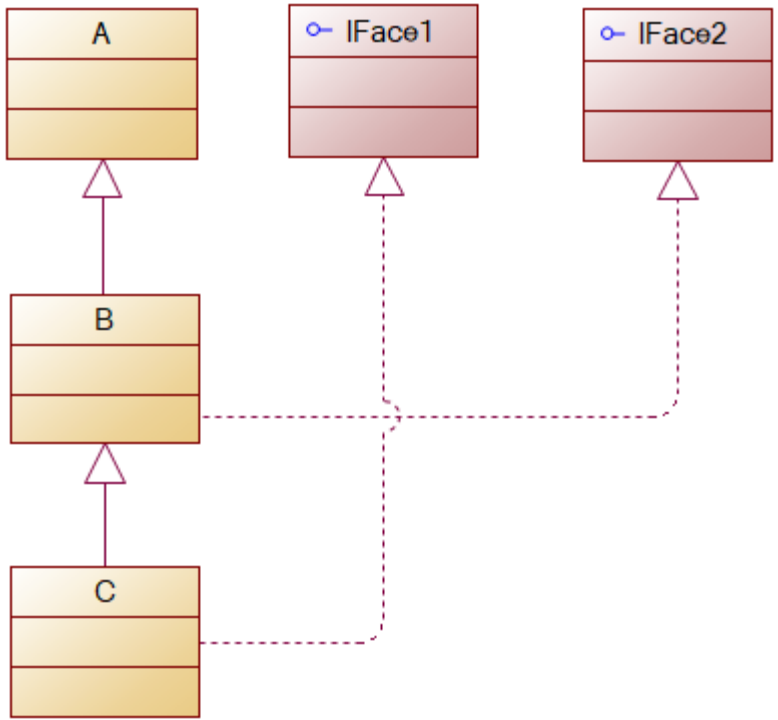
1. Классов.
2. Интерфейсов.

Наследование обозначается символом «:», следующего после имени класса с указанием всех типов, перечисленных через «,», от которых он наследуется.

Класс может наследоваться только от одного класса, и от любого количества интерфейсов.

```
class {className}
                : {abstract classNameParent}
                , {interfaceName1}
                …
                , {interfaceNameN}
{
    //Реализация интерфейсных методов
}
```

Пример синтаксиса

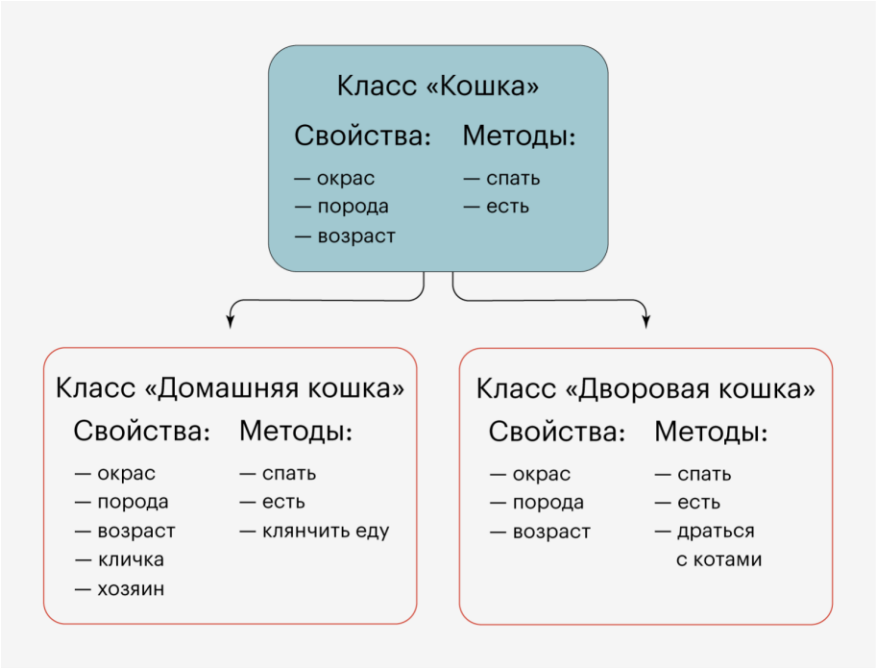



```
abstract class A
{
    //код класса
}
interface IFace1
{
    //код интерфейса
}
interface IFace2
{
    //код интерфейса
}
class B : A, IFace2
{
    //код класса
}
class C : B, IFace1
{
    //код класса
}
```

Что наследуется

В C#, при наследовании у родительского (базового) класса наследуются все **публичные** и **защищенные** (protected) члены класса, включая:

1. **Поля** (если они имеют protected или public модификаторы).
2. **Методы** (с protected или public модификаторами).
3. **Свойства** (с protected или public модификаторами).



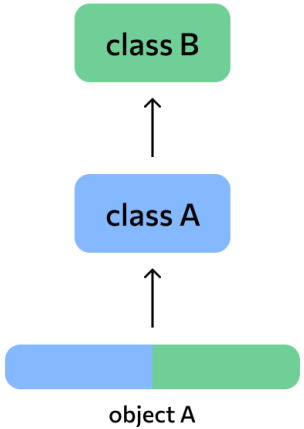
Некоторые нюансы наследования

В рамках .Net классами **не наследуются**:
Классы и методы, помеченные ключевым словом sealed(запечатанный).

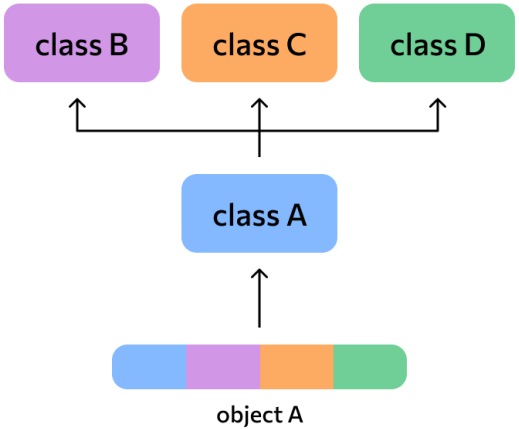
Также есть нюансы, о которых не стоит забывать

1. **Конструкторы**. Код конструктора класса-потомка будет воспроизводить код класса-предка. При этом сначала будет выполняться код класса-предка, потом – класса-потомка.
2. **Статичные члены (static).** Статичные члены класса будут унаследовать поведение класса-предка.
3. **Закрытые члены (private)**. К приватным полям и методам класса напрямую обращаться нельзя. При этом допустимо использовать методы для работы с данными членами.

Множественное наследование запрещено

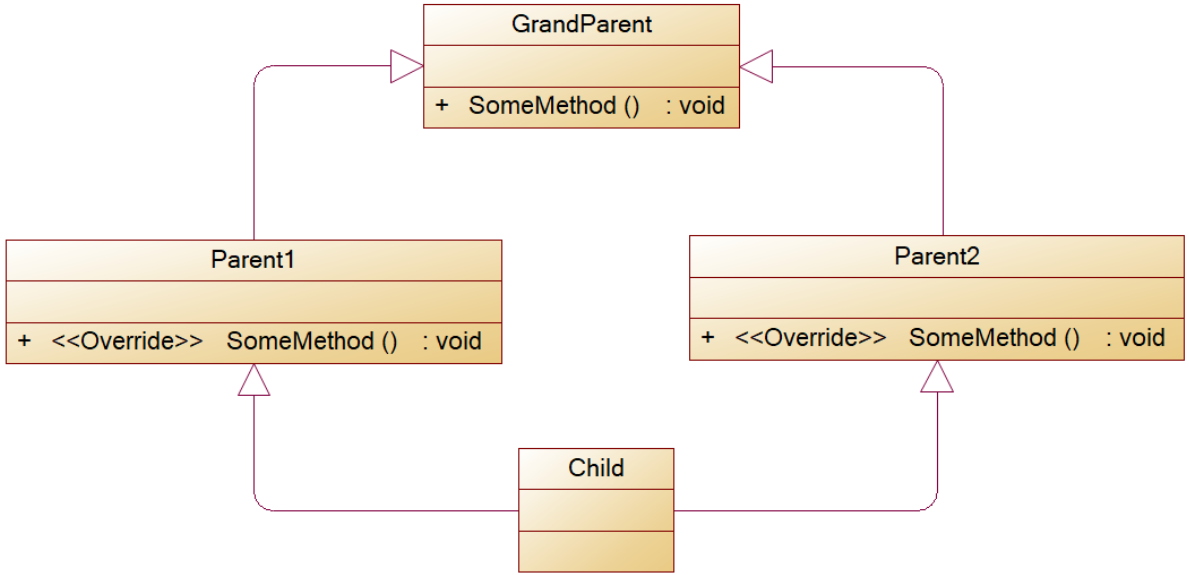


Одиночное наследование



Множественное наследование

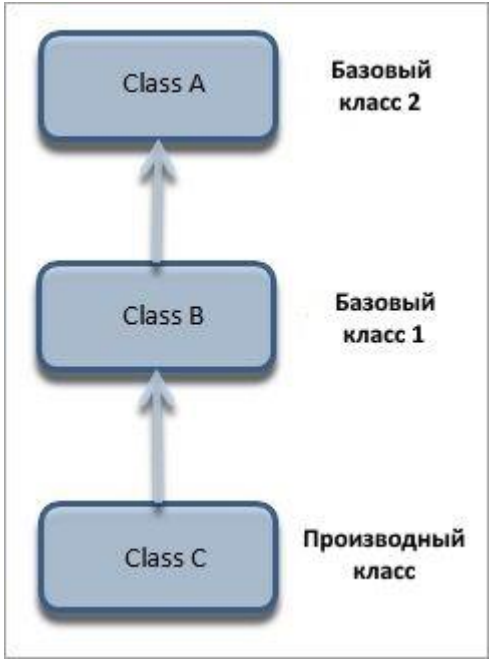
Почему в C# множественное наследование запрещено



Реализация какого класса метода **`SomeMethod`** достанется классу **`Child?`**

Транзитивное наследование

Транзитивное наследование позволяет определить иерархию наследования для набора типов. Другими словами, тип C может наследовать возможности типа B, который в свою очередь наследует от типа A. Благодаря транзитивности наследования члены типа A будут доступны для типа C.



Ключевое слово base

Ключевое слово base используется для доступа к членам базового класса из производного класса.
Используйте его, если вы хотите:

1. Вызвать метод базового класса.
2. Определить конструктор базового класса, который должен вызываться при создании экземпляров производного класса.

```
public class BaseClass
{
    protected void GetInfo()
    { }

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }
}
public class DerivedClass : BaseClass
{
    private void SomeMethod()
    {
        base.GetInfo();
    }
    public DerivedClass() : base()
    { }
}
```

Ключевое слово virtual

Ключевое слово virtual может применяться для:

1. Методов
2. Свойств
3. Событий

virtual позволяет иметь реализацию в методе по умолчанию и переопределять ее в дочерних классах.

```
class Animal
{
    private string name = "Animal";
    public virtual string Name
    {
        get { return name; }
        set { this.name = value; }
    }
    public virtual void Speak()
    { Console.WriteLine("Animal makes a sound"); }
}
class Dog : Animal
{
    public override string Name { get; set; }
    public override void Speak()
    { Console.WriteLine("Dog barks"); }
}
```

Ключевое слово override

Ключевое слово override в C# используется для переопределения виртуальных методов и свойств объявленных в базовом классе.

Оно позволяет изменить поведение базового члена в производном классе, сохраняя полиморфизм.

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal makes a sound");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}
```

Вопросы, на которые отвечает наследование

Итого:

1. Какими свойствами должен обладать объект, созданный из класса-наследника.
2. Какое поведение будет реализовывать объект, созданный из класса наследника.

Решение задач

1 Описать класса-родителя и класс наследника
1.1 Описать класс Person
1.1.1 Персона содержит поле "имя".
1.1.2 Персона содержит не переопределяемый метод, который возвращает значение поля.
1.1.3 Персона содержит переопределяемый метод SayAboutYourSelf(), в котором сообщает свое имя. (virtual)
1.1.4 Персона содержит конструктор, который принимает 1 параметр - имя.
1.2 Описать класс Employee
1.2.1 Работник содержит поле "Компания".
1.2.2 Работник содержит переопределяемый метод DoWork().
1.2.3 Работник переопределяет метод SayAboutYourSelf(), в котором сообщает свое имя и говорит на каком предприятии работает. (override)
1.2.4 Работник содержит конструктор, который принимает имя и название компании как параметры. (использовать base)

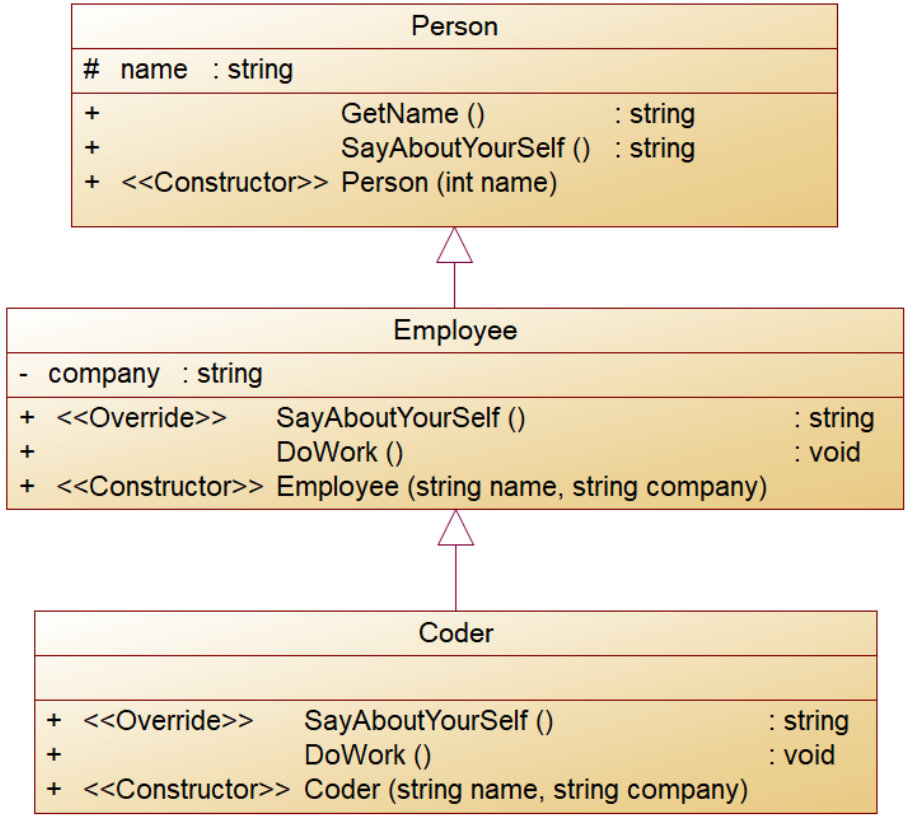
Решение задач

1.3 Описать класс Coder

1.3.1 Программист содержит конструктор, переопределяемый Employee с теми же параметрами (base)

1.3.2 Программист пишет код в методе DoWork() (override)

1.3.3 Программист содержит переопределяемый метод SayAboutYourSelf(), в котором сообщает то же, что и работник + говорит на каком языке пишет.(override + base)

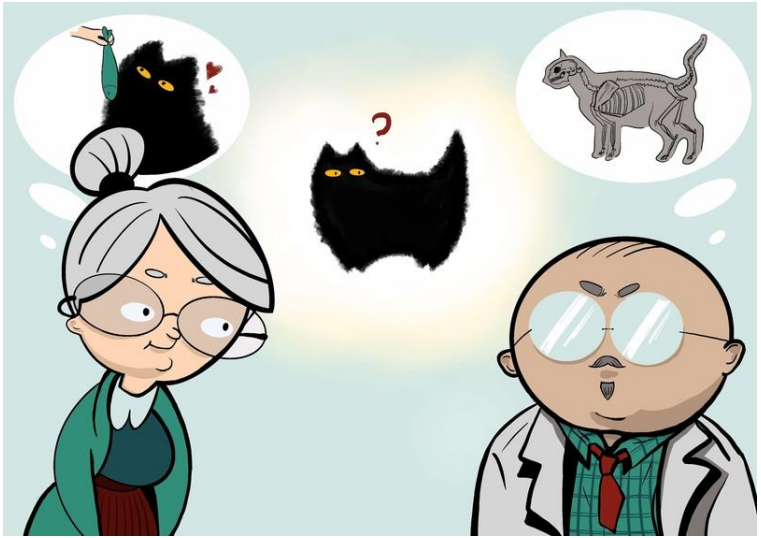


Абстракция

Что такое абстракция?

Абстракция в объектно-ориентированном программировании (ООП) представляет собой концепцию, которая позволяет выделить важные характеристики и свойства объектов, игнорируя при этом ненужные детали.

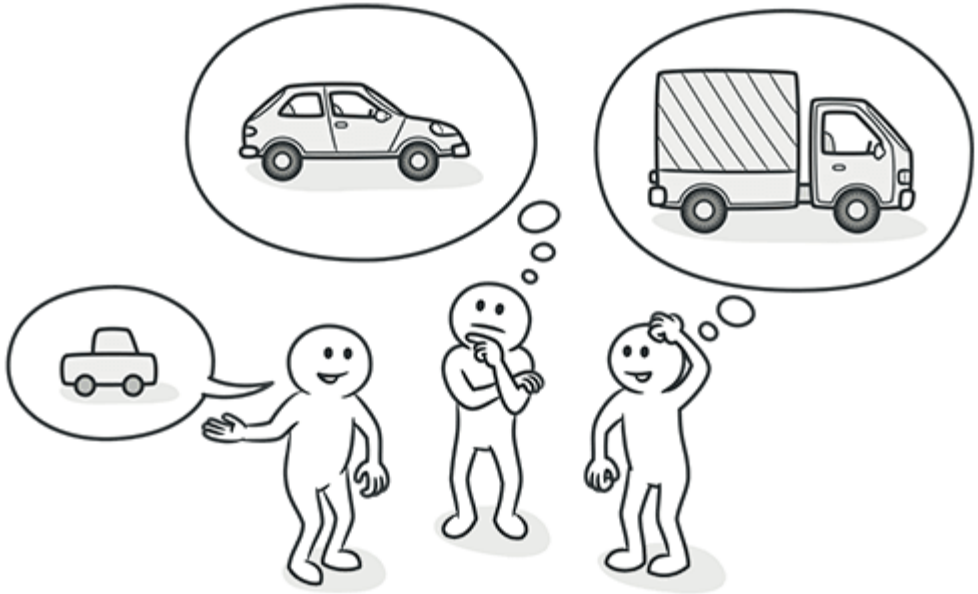
Это позволяет разработчикам сосредотачиваться на ключевых аспектах системы, не углубляясь во все технические детали реализации.



Суть абстракции

Основная идея состоит в том, чтобы представить объект обладающим набором методов и при этом не предоставлять конкретную логику этих методов

Важным аспектом абстракции является то, что нельзя создать объект абстрактного типа, так как он допускает неопределенное поведение.



Представление абстракции в C#

Абстракция в рамках C# представлена двумя сущностями:
1.Абстрактные классы
2.Интерфейсы

Чаще как абстракцию используют интерфейсы, так как наследоваться можно от любого их количества.

Абстрактные классы используют реже, но наследоваться можно только от одного класса, в т.ч. абстрактного.

```
abstract class AbstractAnimal
{
    public abstract void Eat();
}

interface IAnimal
{
    void MakeSound();
}
```

Ключевое слово abstract

Ключевое слово abstract может применяться для:

Абстрактный класс:
1. Нельзя создать объект абстрактного класса.
2. Может содержать как абстрактные методы (без реализации), так и обычные (с реализацией).

Абстрактный метод:
1. Определяется только в абстрактных классах.
2. Не имеет реализации в базовом классе (только заголовок метода).
3. Должен быть переопределён в каждом производном классе.

```
public abstract class Animal
{
    private string name;

    public string GetName()
    {
        return this.name;
    }

    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
       Console.WriteLine("Dog barks");
    }
}
```

Ключевое слово override

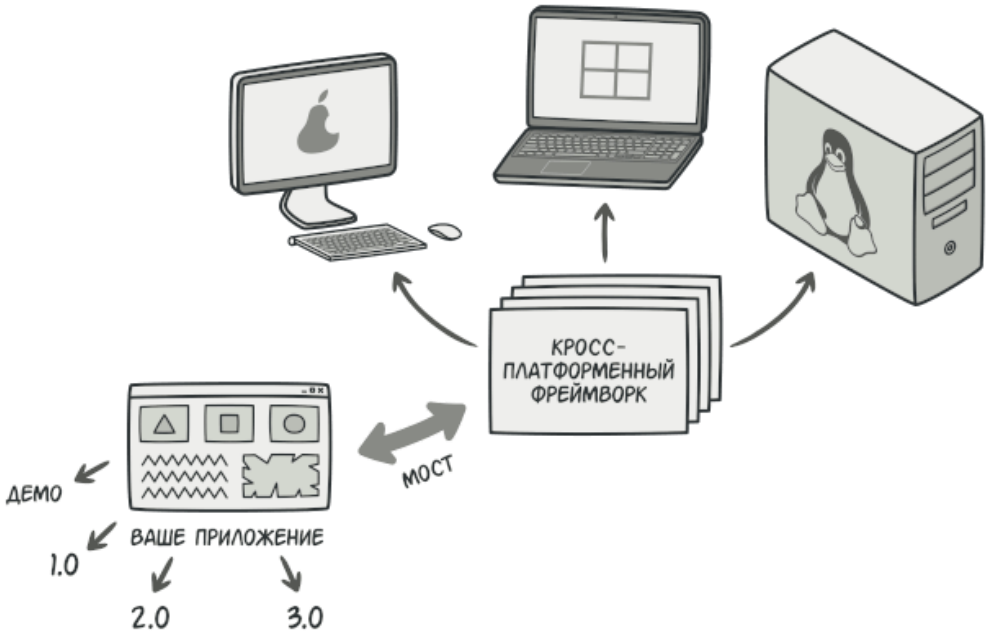
Ключевое слово override в C# используется для переопределения абстрактных методов и свойств объявленных в классе.

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal makes a sound");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}
```

Пример на паттерне «Мост»

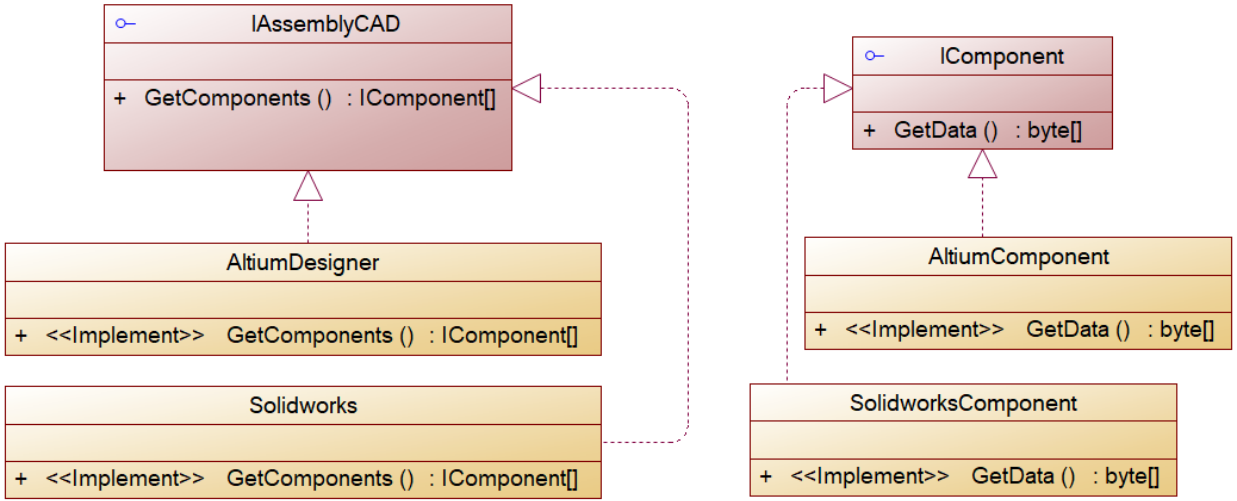
Абстракция будет делегировать работу одному из объектов реализаций. Причём, реализации можно будет взаимозаменять, но только при условии, что все они будут следовать общему интерфейсу.

Таким образом, вы сможете изменять графический интерфейс приложения, не трогая низкоуровневый код работы с операционной системой.



Пример из практики ☺

На предприятии требовалось реализовать получение одних и тех же данных из разных CAD-систем. Данные на выходе имели один и тот же вид. Если реализовать интерфейсы под любой CAD, то результат был бы такой же.



Вопросы, на которые отвечает абстракция

Итого:

1. Абстракция отвечает на вопрос «что делать?», но абстракция не отвечает на вопрос «как?».
2. Какое поведение должно быть определено у объекта наследуемого типа.

Решение задач

Абстракция.

1. Описать абстрактный класс Animal
1.1 Добавить поле name
1.2 Добавить неабстрактный метод GetName()
1.3 Добавить абстрактный метод MakeSound()
2. Описать интерфейсы
2.1 Описать интерфейс IFlying
2.1.1 Добавить метод Fly(лететь)
2.2 Описать интерфейс IWalking
2.2.1 Добавить метод Walk(ходить)
2.3 Описать интерфейс ISwiming
2.3.1 Добавить метод Swim(плавать)
2.4 При желании добавить свойства: крылья(wings), ноги(legs), (плавники)fins.

Решение задач

3. Описать классы с наследованием от необходимых интерфейсов:
3.1 Cat
3.2 Eagle
3.3 Fish
3.4 Duck
3.5 Создать массив объектов классов из п.3.1 – 3.4

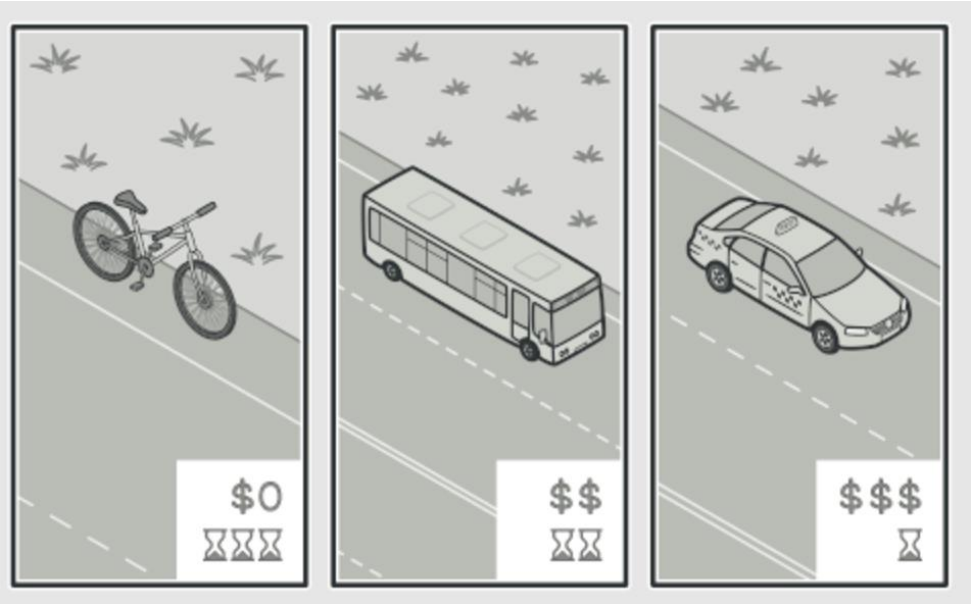
Полиморфизм

Что такое полиморфизм?

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

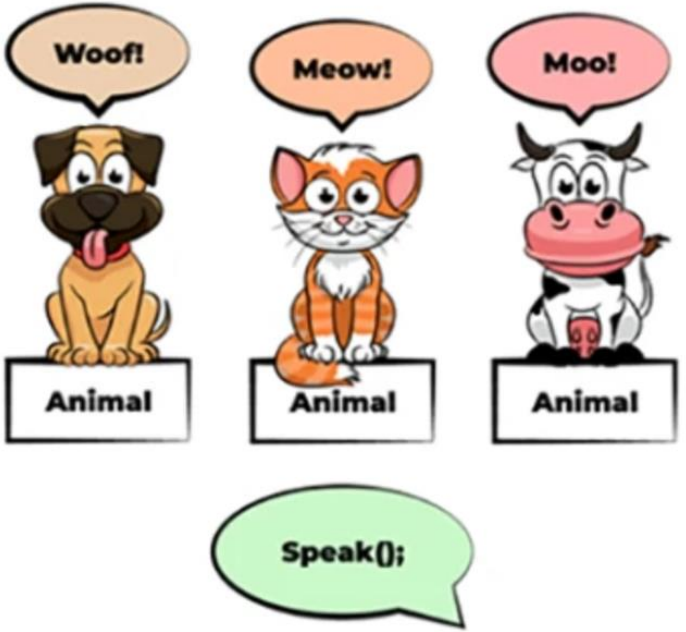
Формы полиморфизма:

1. Полиморфизм подтипов.
2. Параметрический полиморфизм.
3. Полиморфизм ad hoc («специально для этого»)



Полиморфизм подтипов

Полиморфизм подтипов (subtype polymorphism) — это форма полиморфизма, которая возникает, когда один и тот же интерфейс или базовый класс может быть использован для работы с объектами разных типов (подтипов). Это классический вид полиморфизма, связанный с объектно-ориентированным программированием, где подтипы наследуют поведение от базового типа.



Параметрический полиморфизм

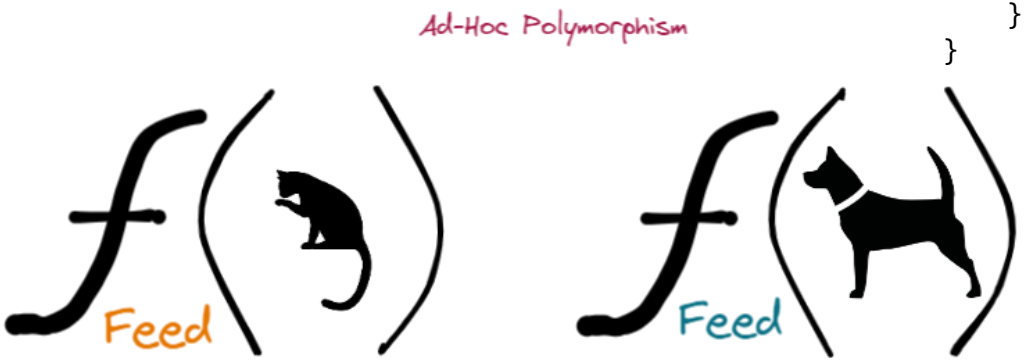
Параметрический полиморфизм — это форма полиморфизма, при которой функции, методы или классы могут работать с разными типами данных, но без привязки к конкретному типу на этапе написания кода. Вместо этого тип задаётся как параметр и определяется при использовании функции или создания экземпляра класса.

```
List<string> strings = new List<string>();

List<int> ints = new List<int>();
```

Ad hoc полиморфизм

Ad hoc полиморфизм (или **специализированный полиморфизм**) — это форма полиморфизма, при которой функции, методы или операторы могут иметь разные реализации в зависимости от типов аргументов, переданных им.

```
class AdHoc
{
    public decimal Add(decimal a, decimal b)
    {
        return a + b;
    }

    public string Add(string a, string b)
    {
        return a + b;
    }
}
```

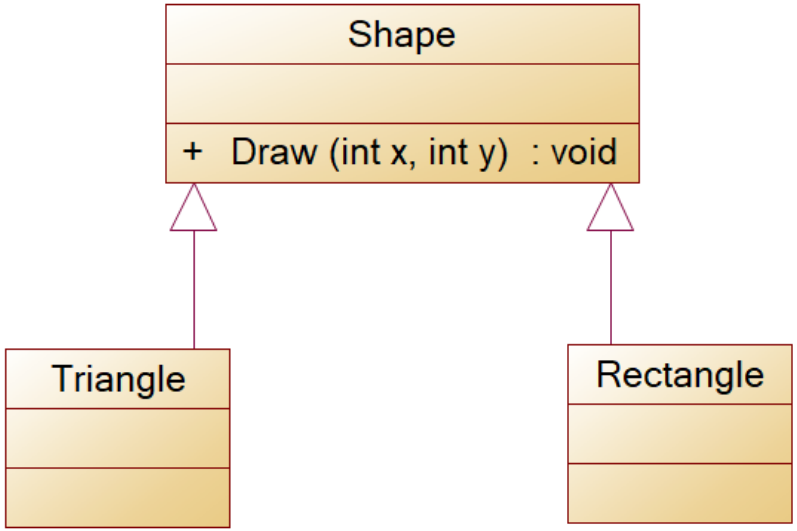


Полиморфизм в действии

Благодаря использованию полиморфизма, мы можем обращаться ко всем объектам, используя один и тот же интерфейс.

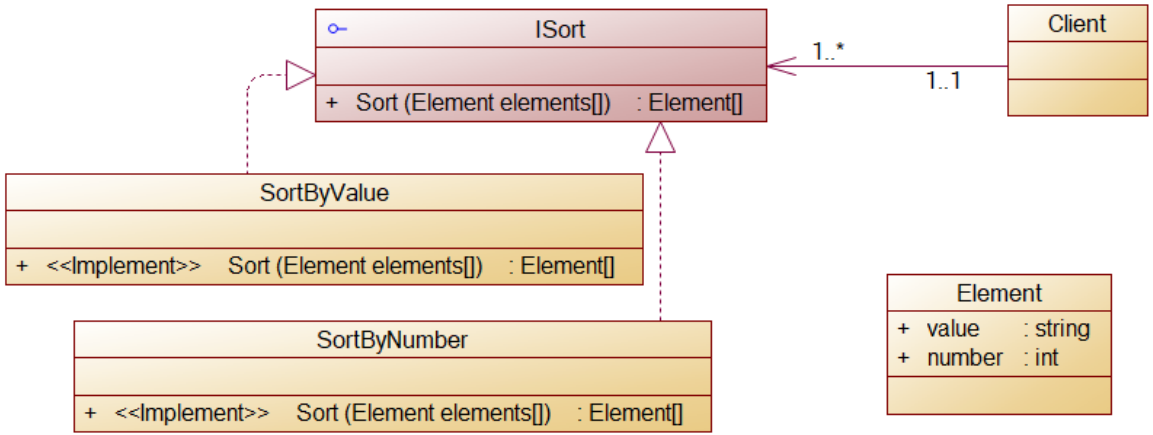
Пример:
Допустим у меня есть массив фигур, которые нужно нарисовать, используя декартовы координаты:

```
foreach (Shape shape in shapes)
    shape.Draw(0, 0);
```



Пример из практики ☺

Требовалась определенная сортировка массивов данных в зависимости от того их содержимого.



Вопросы, на которые отвечает полиморфизм

Итого:

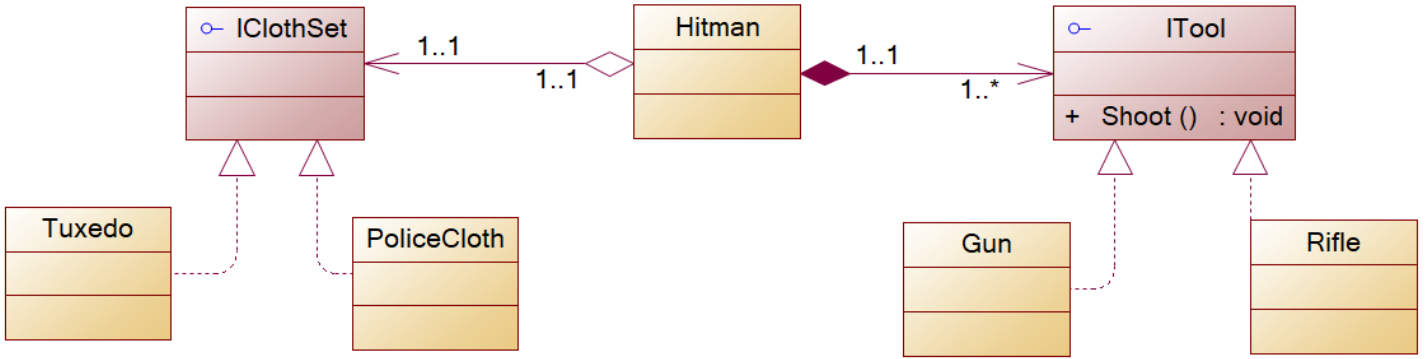
1. Я обращаюсь к объекту, я не зная его внутренней реализации.
2. Код универсальный и сделает, что нужно (при правильной реализации интерфейса).

Задачи по тематике полиморфизма

1. Описать класс Hitman
1.1 Добавить в класс Hitman поле с currentTool типа ITool (интерфейс)
1.2 Добавить в класс Hitman поле и свойство currentClothSet типа IClothSet (интерфейс)
1.3 Добавить метод SetNextTools()
1.4 Добавить метод ChangeClothes(IClothset clothSet)
1.5 Добавить в класс Hitman поле allTools типа List<ITool>. + инициализировать
1.6 Добавить в класс Hitman метод ShootViaCurrentTool()
2. Описать интерфейсы
2.1 Описать интерфейс ITool
2.1.1 Добавить метод Shoot() без реализации
2.1.2 Создать >=2 любых реализаций интерйейса.
2.2 Описать интерфейс IClothset
2.2.1 При желании можно добавить пару методов
2.2.2 Создать >=2 любых реализаций интерйейса.

Задачи по тематике полиморфизма

3. Клиентский код:
3.1 Создать экземпляр класса Hitman
3.2 Сменить оружие на любое другое. Если currentTool пустое, то задать ему начальное значение
3.3 Создать экземпляр одежды
3.4 Сменить комплект одежды на созданный в п.3.3
3.5 Выстрелить текущим оружием

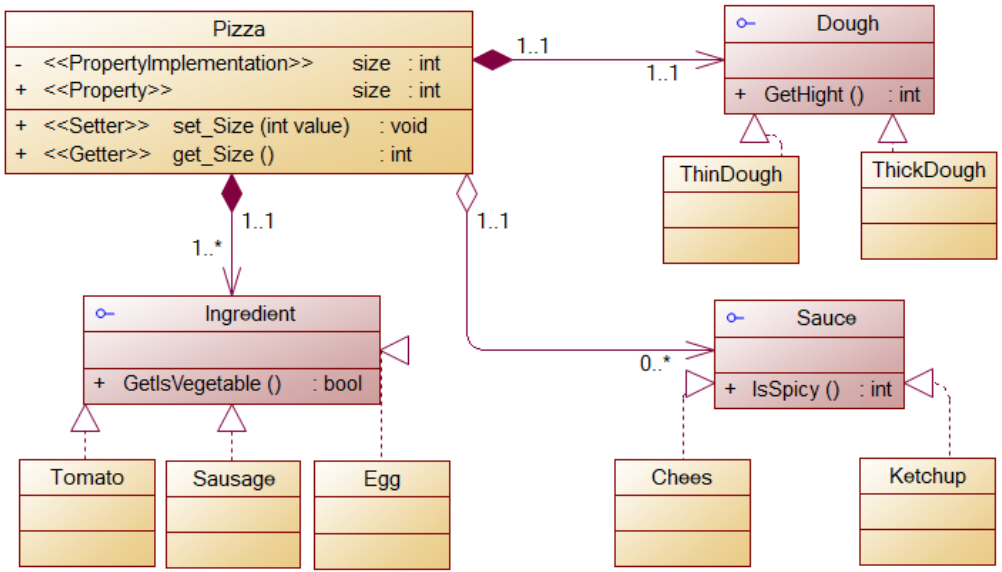


UML (бонус)

Что такое UML

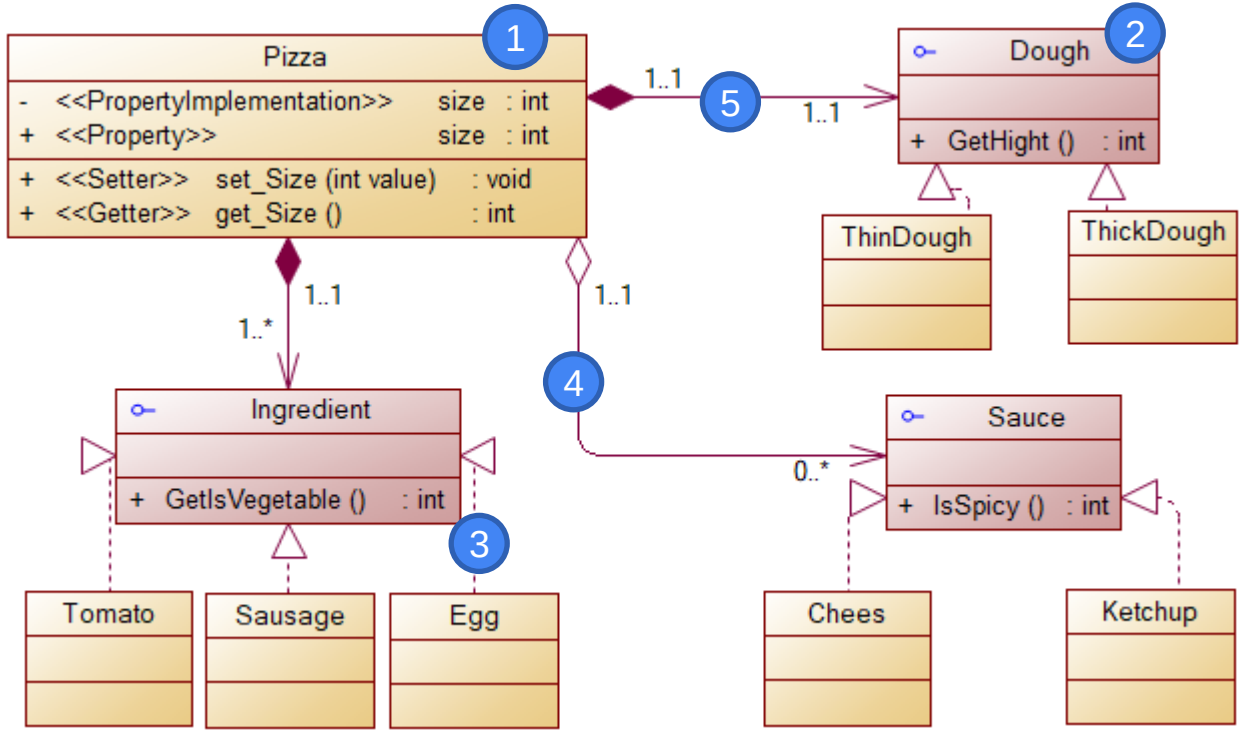
UML, или **Unified Modeling Language**, — это унифицированный язык моделирования. Его используют, чтобы создавать диаграммы и схемы для визуализации процессов и явлений.

Диаграмма классов (англ. *class diagram*) — структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их кооперации, атрибутов (полей), методов, интерфейсов и взаимосвязей (отношений) между ними.



Элементы диаграмм

1. Сущность класса
2. Сущность интерфейса
3. Отношение наследования
4. Отношение агрегации
5. Отношение композиции



Примеры диаграмм в паттернах проектирования

1. Паттерн стратегия
2. Паттерн абстрактная фабрика
3. Паттерн компоновщик

Ответы на вопросы

Рефлексия

Цели вебинара

1. Узнать, что такое ООП.
2. Основы ООП.
3. Особенности синтаксиса ООП.

Вопросы?

Задаем
вопросы в чат

Ставим “–”,
если вопросов нет