

Григорьева Анастасия
Викторовна

Nastyu001@mail.ru

C#

объектно- ориентированное программирование

Санкт-Петербургский Университет,
Математико-Механический Факультет,
2017

План доклада

- Определение понятия «класс»
- 3 столпа ООП (определения)
- Их реализация в С#:
 - Инкапсуляция
 - Наследование
 - Полиморфизм
- Жизненный цикл объектов

Формальное определение класса в C#

- Класс – пользовательский тип данных, который состоит из данных и функций для выполнения действий с этими данными.

К примеру, нам потребовалось создать модель сотрудника некоторой организации

```
class Employee
{
    //Внутренние закрытые данные класса
    private string FullName;
    private int EmpID;
    private float CurrPay;

    //Конструкторы

    public Employee(string FullName, int EmpID, float CurrPay)
    {
        this.FullName = FullName;
        this.EmpID = EmpID;
        this.CurrPay = CurrPay;
    }

    //Метод для увеличения зарплаты сотрудника
    public void GiveBonus(float amount)
    { CurrPay += amount; }
}
```

```
//вызывает конструктор по умолчанию //Ошибка
Employee e = new Employee();
```

Указание области видимости на уровне типа:



Могут быть созданы и из своего двоичного файла и из других двоичных файлов.

(по умолчанию)
Могут быть созданы только из той сборки, в которой были определены.

```
public class Employee  
{  
    ...  
}
```

```
internal class Employee  
{  
    ...  
}
```

Атрибут видимости м.б. установлен для любого типа C#: класса, структуры, перечисления, интерфейса и делегата.

3 важнейших принципа ООП:

- **Инкапсуляция:** как объекты прячут свое внутреннее устройство;
- **Наследование:** как поддерживается повторное использование кода;
- **Полиморфизм:** как реализована поддержка выполнения нужного действия в зависимости от типа передаваемого объекта;

Упрощенно:

- **Инкапсуляция:** способность скрывать детали объекта от пользователя.

«Всё что может быть `private` должно быть `private`»

- **Наследование:** возможность использовать уже созданный (базовый) класс для того чтобы делать расширенные классы. У которых те же атрибуты и методы, что и у базового + свои.

- **Полиморфизм:** (*Poly* = много, *morph* = форма) то, как классы, унаследованные от одного базового, выполняют методы своего класса-родителя.

- Каждый по-своему
- Как в базовом
- Частично как в базовом + особенности и т.д.

Как эти принципы
реализованы в
C#

(Синтаксис)

Инкапсуляция

Модификаторы области видимости:

public

открытый член класса.
Может быть прочитан
или изменен откуда
угодно

protected

защищенный член
класса.

Доступен напрямую из
собственного класса и из
всех дочерних классов.

«семейная тайна»

private

(по умолчанию)
закрытый член класса.
Доступен только из того
объекта, в котором они
определялись

Принцип инкапсуляции предполагает, что ко внутренним данным объекта нельзя обратиться напрямую через экземпляр этого объекта.

Вместо этого для получения информации о состоянии объекта и изменений предлагается использовать специальные методы.

Рассмотрим пример:

Пусть у нас переменная, отвечающая за количество страниц в книге будет открытой, объявлена с ключевым словом **public**.

```
public class Book
{
    public int numberOfPages;
}
```

Тогда возникает проблема: полю можно присвоить любое значение, а организовать проверку этого значения в основном коде достаточно сложно.

Например, тут используется тип данных *int*. Максимальное значение для этого типа - достаточно большое число (2 147 483 647). Если в программе будет такой код, проблем со стороны компилятора не возникнет:

```
class Program
{
    //Задумаемся...
    static void Main(string[] args)
    {
        Book miniNovel = new Book();
        miniNovel.numberOfPages = 30000000;
    }
}
```

А можем и вообще отрицательное число страниц задать...

Чтобы решить проблему нужно сделать это поле закрытым (**private** или **protected**), а для обращения к нему воспользоваться одним из двух способов:

1. Создать традиционную пару методов – один для чтения переменной а второй для её изменения:

```
public class Book
{
    private int numberOfPages;
    public int accessor() { return numberOfPages; }
    public void mutator(int n)
    {
        if ((n > 0) && (n < 2000)) { numberOfPages = n; }
    }
}
```

Теперь:

```
Book miniNovel = new Book();
miniNovel.numberOfPages = 2; //Ошибка компиляции!
```

Но можно так:

```
Book miniNovel = new Book();
miniNovel.mutator(300); //OK
miniNovel.mutator(3000000000); //Тут переменная не изменилась
Console.WriteLine(miniNovel.accessor()); //300
```

2. Определить именованное свойство:

```
private int numberOfPages;

//Свойство для numberOfPages
public int NumberOfPages
{
    get {return numberOfPages;}
    set
    {
        if ((value > 0) && (value < 2000)) { numberOfPages = value; }
    }
}
```

Теперь мы можем обращаться как к обычной, открытой переменной:

```
Book miniNovel = new Book();
miniNovel.NumberOfPages = 78;
Console.WriteLine (miniNovel.NumberOfPages); //78
```

Замечание 1: на самом деле свойства всегда отображаются в «реальные» методы доступа и изменения. Первый начинается с приставки `get_`, второй с `set_`. (`get_NumberOfPages()` и `set_NumberOfPages()`).

Замечание 2: чтобы сделать свойство доступным только для чтения, достаточно пропустить в нем блок `set`

Замечание 3: статическим переменным нужны статические свойства (***public static*** `string Name`).

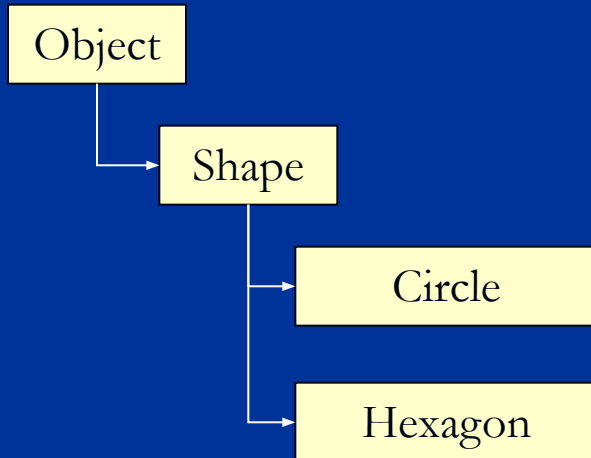
Наследование

Наследование

has-a

(классическое)

Получают функциональность от базового класса-предка и дополняют новыми возможностями

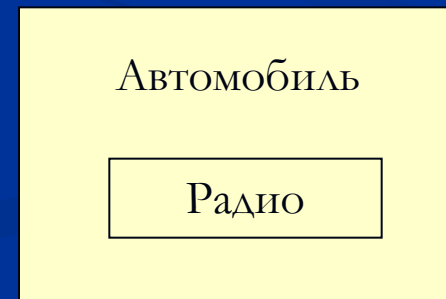


is-a

(включение-

делегирование)

Один класс включает в свой состав другой и открывает внешнему миру часть возможностей внутреннего класса.



Классическое наследование в C#

Определение:

```
class Manager : Employee
{
    //Менеджерам необходимо знать количество
    //имеющихся у них опционов на акции
    private ulong numberOfOptions;
    public ulong NumOpts
    {
        get { return numberOfOptions; }
        set { numberOfOptions = value; }
    }
}

class SalesPerson : Employee
{
    //Продавцам нужно знать объем своих продаж
    private int numberOfSales;
    public int NumSles
    {
        get { return numberOfSales; }
        set { numberOfSales = value; }
    }
}
```

Создаем объект производного класса :

```
static void Main(string[] args)
{
    //Создаем объект "продавец"
    SalesPerson stan = new SalesPerson();

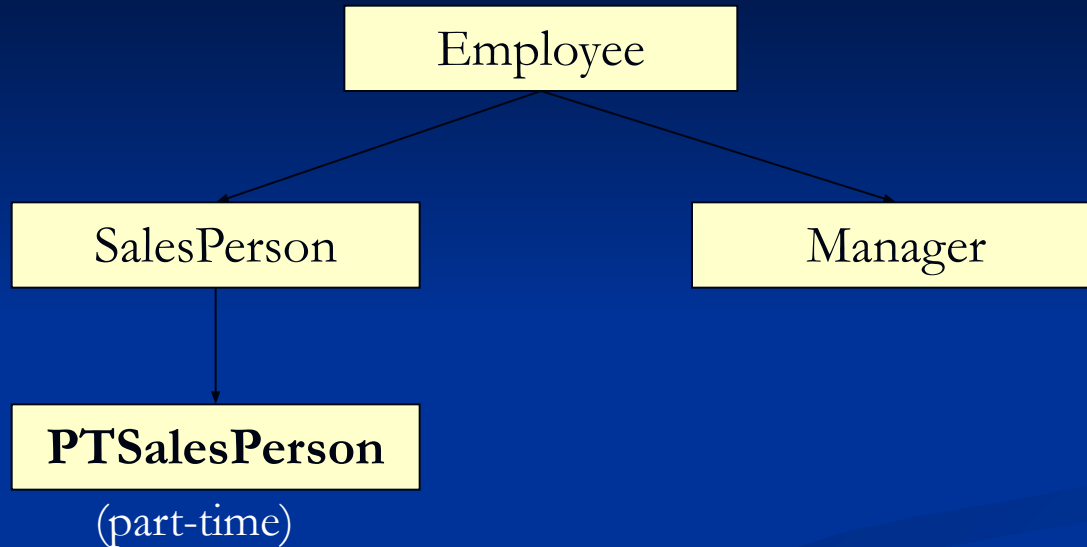
    //Эти члены унаследованы от
    //базового класса Employee
    stan.GiveBonus(100);
    stan.EmpID = 13;

    //А это уникальный член,
    //определенный только в классе
    stan.NumSles = 42;
}
```

Некоторые замечания:

- Указатель на базовый класс «:»
- Наследуются только ОТКРЫТЫЕ и PROTECTED члены базового класса
- Нельзя производить наследование от нескольких базовых классов

Запрет наследования

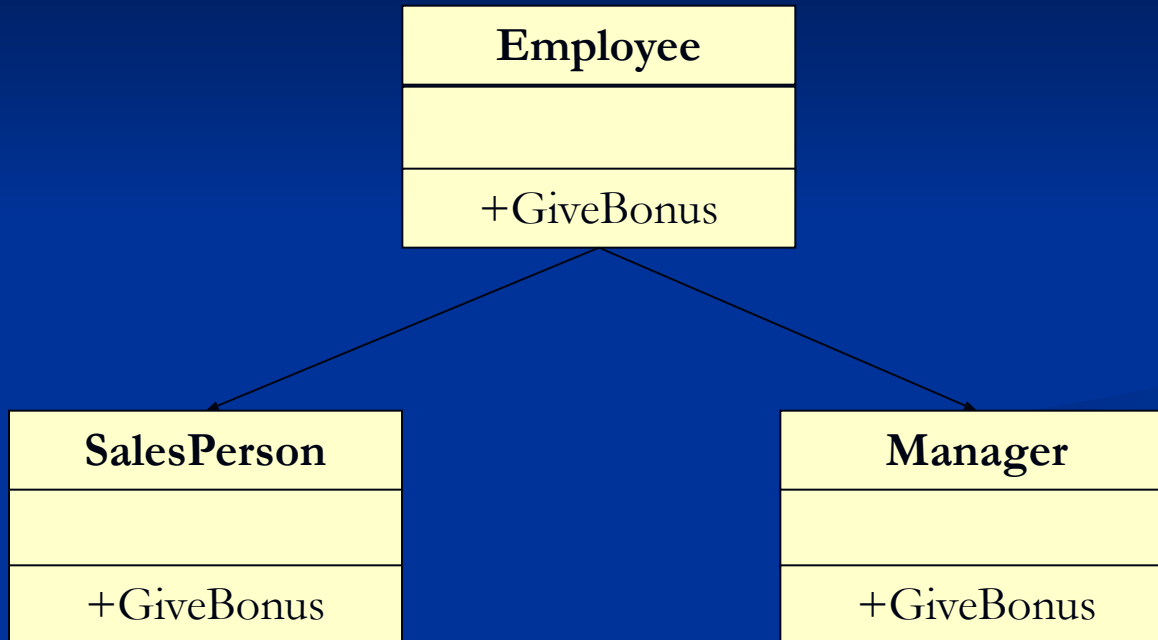


Для запрещения наследования предусмотрено ключевое слово **sealed**

```
//Запрещаем этому классу быть базовым для остальных  
public sealed class PTSalesPerson : SalesPerson  
{  
    //код класса  
}
```

Полиморфизм

Переопределим реакцию объектов производных классов на метод базового класса.



Метод в базовом классе, который будет переопределен должен быть объявлен как виртуальный (ключевое слово **virtual**)

```
class Employee
{
    //Метод для увеличения зарплаты сотрудника
    public virtual void GiveBonus(float amount)
    {
        CurrPay += amount;
    }
}
```

Переопределяя виртуальный метод в производном классе, мы заново определяем его, используя ключевое слово **override**

```
class SalesPerson : Employee
{
    //Влияет объем продаж
    public override void GiveBonu
    {
        int salesBonus = 0;
        if (numberOfSales >= 0 && n
            salesBonus = 10;
        else if (numberOfSales >= 1
            salesBonus = 15;
        else
            salesBonus = 20; //Для
        base.GiveBonus(amount * sal
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //Улучшенная система поощрений
        Manager chucky = new Manager();
        chucky.GiveBonus(300);

        //Создаем объект "продавец"
        SalesPerson stan = new SalesPerson();
        stan.GiveBonus(500);
    }
}
```

```
class Manager : Employee
{
    a();
    ke получают опционы на акции
    is(float amount)

    :.Next(500);
}
```

Если не переопределяем, реализуется как в базовом.

Абстрактные классы и методы

Абстрактный класс:

```
//Объявляем Employee абстрактным, запрещая создание объектов этого класса
abstract public class Employee
{
    //Внутренние данные класса
}
```

Абстрактные методы могут быть в базовом классе без реализации по умолчанию. Каждый производный класс обязан самостоятельно определить этот метод.

```
//Объявляем Employee абстрактным, запрещая создание объектов этого класса
abstract public class Employee
{
    //Абстрактный метод
    public abstract void GiveBonus(float amount);
}
```

Производный класс в котором не будет замещен абстрактный метод сам считается абстрактным, и мы не сможем создавать объекты этого класса.

Жизненный цикл объекта

- **Создание:** используется ключевое слово **new**, обращаемся к конструктору объекта.
- **Уничтожение:** обращаемся к деструктору.

Вопрос: как понять, что объект больше не нужен?

Ответ: Удаляем объект из памяти когда в текущей области видимости на него больше нет активных ссылок.

Удаление делается автоматически средой выполнения .NET

Когда заканчивается место в управляемой куче, запускается сборщик мусора.

Сначала деструктор, потом физическое удаление из памяти.

Вопросы?