История развития ООП. Базовые понятия ООП. Основные принципы ООП

Цели и задачи занятия

Цель занятия: Формирование общего представления и базовых понятий об объектно-ориентированном программировании.

Задачи занятия:

Задачи занятия:

Дидактические:

- познакомить с историей развития ООП
- дать представление о понятиях «объект», «класс»;
- изучить основные принципы ООП;

Развивающие:

- развить навыки сравнения ;
- формировать умения анализировать и сопоставлять;

Воспитательные:

- *Формировать интерес к профессии, дисциплинированность;*

Немного истории

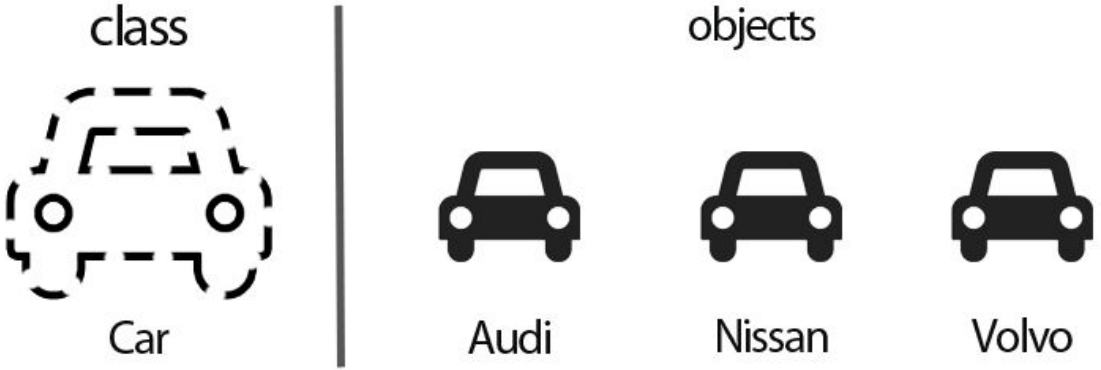
- Первым языком программирования, в котором были предложены основные понятия ООП, была **Си́мула-67 (Simula 67)**. В момент его появления в 1967 году в нём были предложены революционные идеи: **объекты**, **классы**, **виртуальные методы** и др.

- Первым широко распространённым объектно-ориентированным языком программирования стал **Smalltalk**. Здесь понятие **класса** стало основообразующей идеей для всех остальных конструкций языка.

Так что же такое ООП?

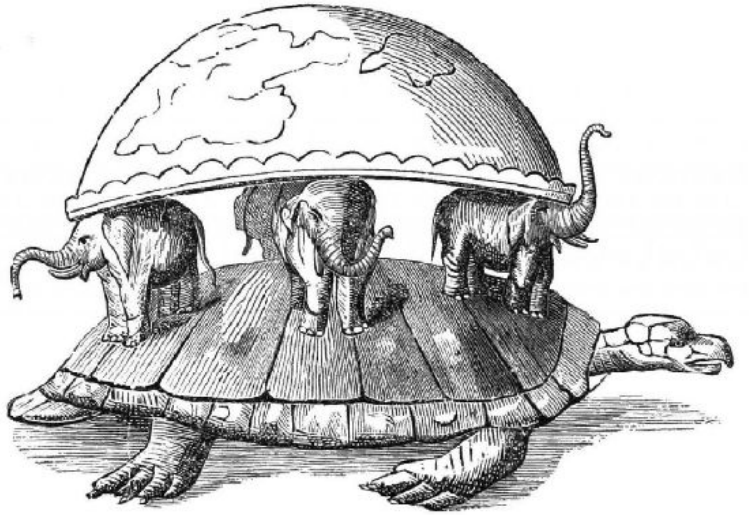
Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Понятия класс и объект



Класс – это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д. Описав класс, мы можем создать его экземпляр – **объект**. Объект – это уже конкретный представитель класса.

Принципы ООП



1. Инкапсуляция
2. Абстрагирование
3. Наследование
4. Полиморфизм

Принцип 1. Инкапсуляция

Инкапсуляция – это скрытие реализации объекта от конечного пользователя, которое в C# осуществляется при помощи модификаторов доступа. Конечным пользователем объекта здесь выступает либо объект наследник, либо программист.

Изначальное значение слова «инкапсуляция» в программировании — объединение данных и методов работы с этими данными в одной упаковке («капсуле»).

Принцип 1. Инкапсуляция

public	доступ к члену возможен из любого места одной сборки, либо из другой сборки, на которую есть ссылка;
protected	доступ к члену возможен только внутри класса, либо в классе-наследнике (при наследовании);
internal	доступ к члену возможен только из сборки, в которой он объявлен;
private	доступ к члену возможен только внутри класса;
protected internal	доступ к члену возможен из одной сборки, либо из класса-наследника другой сборки.

Пример инкапсуляции

```
using System;

namespace RectangleApplication {
   class Rectangle {
      //member variables
      public double length;
      public double width;

      public double GetArea() {
         return length * width;
      }
      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
}
```

```
Length: 4.5
Width: 3.5
Area: 15.75
```

Принцип 1. Инкапсуляция / Преимущества

- Контроль за корректным состоянием объекта.

- Удобство для пользователя за счет интерфейса. Мы оставляем «снаружи» для доступа пользователя только методы. Ему достаточно вызвать их, чтобы получить результат, и совсем не нужно вникать в детали их работы.

- Изменения в коде не отражаются на пользователях. Все изменения мы проводим внутри методов. На пользователя это не повлияет: если мы меняем что-то в работе метода, для него останется незаметным: он, как и раньше, просто будет получать нужный результат.

Принцип 2. Абстрагирование

Абстрагирование – позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект.

Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т.д.

Принцип 2. Абстрагирование

!!!

При создании программы также очень важно помнить, что с точки зрения разных задач один и тот же объект может обладать совершенно разными характеристиками.

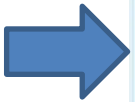
Принцип 3. Наследование

Наследование – позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником.

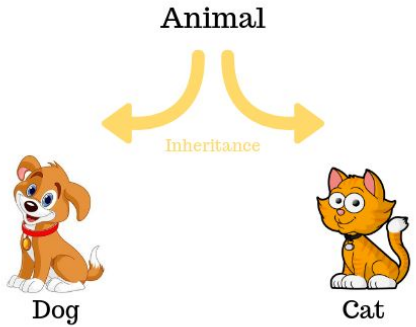
Например, есть базовый класс животное. В нем описаны общие характеристики для всех животных (класс животного, вес). На базе этого класса можно создать классы наследники Собака, Слон со своими специфическими свойствами. Все свойства и методы базового класса при наследовании переходят в класс наследник.

Принцип 3. Наследование

```
class [имя_класса] : [имя_базового_класса]
{
  // тело класса
}
```



```
class Animal
{
  public string Name { get; set; }
}
class Dog : Animal
{
  public void Guard()
  {
    // собака охраняет
  }
}
class Cat : Animal
{
  public void CatchMouse()
  {
    // кошка ловит мышь
  }
}
class Program
{
 static void Main(string[] args)
  {
    Dog dog1 = new Dog();
    dog1.Name = "Барбос"; // называем пса
    Cat cat1 = new Cat();
    cat1.Name = "Барсик"; // называем кота
    dog1.Guard(); // отправляем пса охранять
    cat1.CatchMouse(); // отправляем кота на охоту
  }
}
```



Принцип 3. Наследование/ Преимущества

- ✔ эффективное построение сложных иерархий классов .
- ✔ повторное использование ранее написанного кода
- ✔ удобство в сопровождении
- ✔ уменьшение количества логических ошибок
- ✔ легкость в согласовании разных частей программного кода путем использования интерфейсов.
- ✔ создание библиотек кода, которые можно использовать и дополнять собственными разработками;
- ✔ возможность реализовывать известные шаблоны проектирования
- ✔ использование преимуществ полиморфизма
- ✔ обеспечение исследовательского программирования (быстрого макетирования).
- ✔ лучшее понимание структуры программной системы

Принцип 4. Полиморфизм

Полиморфизмом это способность объектов с одним интерфейсом иметь различную реализацию.

Например, есть два класса, Круг и Квадрат. У обоих классов есть метод GetSquare(), который считает и возвращает площадь. Но площадь круга и квадрата вычисляется по-разному, соответственно, реализация одного и того же метода различная.

«Один интерфейс — много реализаций».

Принцип 4. Полиморфизм

Виртуальный метод – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе.

Абстрактный метод – это метод, который ДОЛЖЕН быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое), в отличии от виртуального.

Переопределение метода – это изменение реализации метода, установленного как виртуальный (в классе наследнике метод будет работать отлично от базового класса).

Принцип 4. Полиморфизм

```
[модификатор доступа] virtual [тип] [имя метода] ([аргументы])
{
  // тело метода
}
```

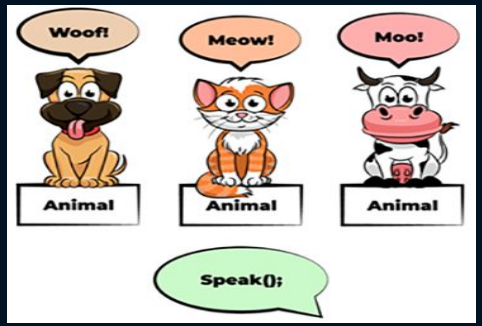
**Статический метод не может быть виртуальным.*

```
[модификатор доступа] override [тип] [имя метода] ([аргументы])
{
  // новое тело метода
}
```

Принцип 4. Полиморфизм
Виртуальные методы в C#. Переопределение методов

```
class Person
{
  public string Name { get; set; }
  public int Age { get; set; }

  public Person(string name, int age)
  {
    Name = name;
    Age = age;
  }
  public virtual void ShowInfo() //объявление виртуального метода
  {
    Console.WriteLine("Человек\nИмя: " + Name + "\n" + "Возраст: " + Age +
"\n");
  }
}
class Student : Person
{
  public string HighSchoolName { get; set; }

  public Student(string name, int age, string hsName)
: base(name, age)
  {
    HighSchoolName = hsName;
  }
  public override void ShowInfo() // переопределение метода
  {
    Console.WriteLine("Студент\nИмя: " + Name + "\n" + "Возраст: " + Age
+"\n"+ "Название ВУЗа: " + HighSchoolName + "\n");
  }
}
class Pupil : Person
{
  public string Form { get; set; }

  public Pupil(string name, int age, string form)
: base(name, age)
  {
    Form = form;
  }
  public override void ShowInfo() // переопределение метода
  {
    Console.WriteLine("Ученик(ца)\nИмя: " + Name + "\n" + "Возраст: " + Age
+ "\n" + "Класс: " + Form + "\n");
  }
}

class Program
{

  static void Main(string[] args)
  {
    List<Person> persons = new List<Person>();
    persons.Add(new Person("Василий", 32));
    persons.Add(new Student("Андрей", 21, "МГУ"));
    persons.Add(new Pupil("Елена", 12, "7-Б"));

    foreach (Person p in persons)
      p.ShowInfo();

    Console.ReadKey();
  }
}
```

```
foreach (Person p in persons)
{
  if (p is Student)
    ((Student)p).ShowInfo();
  else if (p is Pupil)
    ((Pupil)p).ShowInfo();
  else p.ShowInfo();
}
```



Принцип 4. Полиморфизм / Преимущества

- Позволяет записывать методы лишь однажды и затем повторно их использовать для различных типов данных, которые, возможно, еще не существуют (обобщенные действия или алгоритмы).
- Возможность работать с несколькими типами так, как будто это один и тот же тип.
- Один и тот же интерфейс может быть использован для создания методов с разными реализациями.

Поля и методы класса

- **Для хранения данных в классе применяются поля**
- **Для определения поведения в классе применяются методы**

Пример

```
class Person
{
    public string name = "Undefined";    // имя
    public int age;                       // возраст

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Создание объекта класса

Для создания объекта применяются конструкторы.

Синтаксис:

new конструктор_класса (параметры_конструктора);

Пример

```
Person tom = new Person();  // создание объекта класса Person

// определение класса Person
class Person
{
    public string name = "Undefined";
    public int age;

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Обращение к функциональности класса

- объект.поле_класса
- объект.метод_класса (параметры_метода)

Пример

```
Person tom = new Person();  // создание объекта класса Person

// Получаем значение полей в переменные
string personName = tom.name;
int personAge = tom.age;
Console.WriteLine($"Имя: {personName}  Возраст {personAge}");   // Имя: Undefined  Возраст: 0

// устанавливаем новые значения полей
tom.name = "Tom";
tom.age = 37;

// обращаемся к методу Print
tom.Print();    // Имя: Tom  Возраст: 37

class Person
{
    public string name = "Undefined";
    public int age;

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

ВОПРОСЫ

```
class Person
{
    public string name = "Sam";
    public int age;

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

- *слово **this** позволяет обращаться к любому полю или методу.*

Какое значение поле name будет иметь при выполнение следующего кода и почему?

Person tom = new Person("Tom", 34) { name = "Bob", age = 29 };

```
class Person
{
    public string name = "Ben";
    public int age = 18;
    public string email = "ben@gmail.com";

    public Person(string name)
    {
        this.name = name;
    }
    public Person(string name, int age) : this(name)
    {
        this.age = age;
    }
    public Person(string name, int age, string email) : this("Bob", age)
    {
        this.email = email;
    }
}
```

Какое значения будут иметь поля name, age и email после выполнения следующего кода и почему? В каком порядке будут вызываться конструкторы класса Person?

Person person = new Person("Tom", 31, "tom@gmail.com");

Домашнее задание

- Выучить конспект
- Проанализировать программы, которые вы писали до этого, понять правильно ли вы там использовали модификаторы доступа
- Создайте базовый класс *Геометрическая фигура*, предусмотрите в нем общие поля/свойства, например координаты центра фигуры, с помощью конструктора должна быть возможность задать центр. На базе этого класса создайте два новых – *Треугольник* и *Окружность*. В этих классах должны быть свои особые поля, например радиус для окружности. В оба класса добавьте метод *Нарисовать*, в котором могла бы быть специфическая логика рисования фигуры. Создайте объекты треугольник и окружность.

Конструкторы, инициализаторы и деконструкторы

- конструктор представляет метод, который называется по имени класса, имеет параметры, определять возвращаемый тип не надо.
- конструктор выполняет инициализацию объекта
- если в классе определяются свои конструкторы, то он лишается конструктора по умолчанию

```
Person tom = new Person();  // Создание объекта класса Person

tom.Print();    // Имя: Tom  Возраст: 37

class Person
{
    public string name;
    public int age;
    public Person()
    {
        Console.WriteLine("Создание объекта Person");
        name = "Tom";
        age = 37;
    }
    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
```

Создание нескольких конструкторов

```
public Person() { name = "Неизвестно"; age = 18; }
public Person(string n) { name = n; age = 18; }
public Person(string n, int a) { name = n; age = a; }
```

```
Person tom = new Person();
Person bob = new Person("Bob");
Person sam = new Person("Sam", 25);
```

Ключевое слово this

- Ключевое слово this представляет ссылку на текущий экземпляр/объект класса.
- this.name = name; - если параметры и поля называются одинаково, их можно разграничить.
- первая часть - this.name означает, что name - это поле текущего класса, а не название параметра name.
- слово **this** позволяет обращаться к любому полю или методу.

Инициализаторы

Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта.

```
Person tom = new Person { name = "Tom", age = 31 };
```

Инициализаторы

- С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания.
- С помощью инициализатора мы можем установить значения только доступных из вне класса полей и свойств объекта.
- Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Деконструкторы

- Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.

Пример

```
class Person
{
    string name;
    int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public void Deconstruct(out string personName, out int personAge)
    {
        personName = name;
        personAge = age;
    }
}
```

```
Person person = new Person("Tom", 33);

(string name, int age) = person;

Console.WriteLine(name);    // Tom
Console.WriteLine(age);     // 33
```

При получении значений из декоструктора нам необходимо предоставить столько переменных, сколько деконструктор возвращает значений.
Если какие-то значения не нужны, использовать прочерк _

Деструкторы в C#

- Деструкторы в C# — это методы внутри класса, используемые для уничтожения экземпляров этого класса , когда они больше не нужны.

Особенности

- Деструктор уникален для своего класса, т.е. в классе не может быть более одного деструктора.
- Деструктор не имеет возвращаемого типа и имеет то же имя, что и имя класса (включая тот же случай).
- Он отличается от конструктора символом тильды (~) перед его именем.
- Деструктор не принимает никаких параметров и модификаторов.
- Его нельзя определить в структурах. Он используется только с классами.
- Он не может быть перегружен или унаследован.
- Он вызывается при выходе из программы.
- Внутри Destructor вызывается метод Finalize для базового класса объекта.

Синтаксис

```
пример класса
{
    // Остальная часть класса
    // члены и методы.

   // Деструктор
   ~Пример()
    {
        // Ваш код
    }
```

Структуры

Структура (struct) в C# — это пользовательский тип данных, который используется наряду с классами и может содержать какие-либо данные и методы.

Отличие от класса

Основное отличие структуры (struct) от класса (class) заключается в том, что структура — это тип значений, а класс — это ссылочный тип.

Пример

```
public struct Point3D
{
public double X { get; set; }
public double Y { get; set; }
public double Z{ get; set; }
public override string ToString()
{
return $"({X},{Y},{Z})";
}
}
```

Создание структуры

Point3D Point = new Point3D();

Если структура содержит **только публичные поля** (не путать со свойствами) и методы, то можно не вызывать конструктор, а сразу назначить значение полей и после этого вызывать методы структуру.

```
public struct Point3D
{
public double X;
public double Y;
public double Z;
public override string ToString()
{
return $"({X},{Y},{Z})";
}
};
//не вызываем конструктор, а сразу задаем значение полей
Point3D Point;
Point.X = 100;
Point.Y = 100;
Point.Z = 100;
Console.WriteLine(Point.ToString());
```

Конструкторы структур

```
public struct Point3D
{
public double X = 10;
public double Y = 5;
public double Z = 8;
public override string ToString()
{
return $"({X},{Y},{Z})";
}
public Point3D(double x) : this()
{
X = x;
Y = 0;
Z = 0;
}
public Point3D(double x, double y) : this(x)
{
Y = y;
Z = 0;
}
public Point3D(double x, double y, double z) : this(x, y)
{
Z = z;
}
}
```

Инициализатор структур struct

- Point3D Point4 = new() { X = 24, Y = 45, Z = 22 };
- Point3D Point4 = new(10, 20, 30) { X = 24, Y = 45, Z = 22 };

Структуры не поддерживают наследование

В отличие от классов C#, наследование структур не поддерживается, то есть вот такой код приведет к ошибке:

```
struct Point3DType2 : Point3D
{ }
```

СПАСИБО
ЗА ВНИМАНИЕ!