# Классы и объекты

ООП. Объектно-ориентировочное программирование — методология программирования, основанная на представлении программы в виде совокупности [объектов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), каждый из которых является экземпляром определённого [класса](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), а классы образуют иерархию наследования © Wiki.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

**абстрагирование** для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;

**инкапсуляция** для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;

**наследование** для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;

**полиморфизм** для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

В центре ООП находится понятие объекта. Объект — это сущность, которой можно посылать сообщения и которая может на них реагировать, используя свои данные.

#### Немного теории:

Объектно-ориентированное программирование (в дальнейшем ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

В центре ООП находится понятие объекта.

**Объект** — это сущность, экземпляр класса, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Данные объекта скрыты от остальной программы. Сокрытие данных называется инкапсуляцией.

Наличие инкапсуляции достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности — для этого требуется наличие наследования.

Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП. Основные преимущества ООП проявляются только в том случае, когда в языке программирования реализован полиморфизм, то есть возможность объектов с одинаковой спецификацией иметь различную реализацию.

Хочу выделить что очень часто натыкаюсь на мнение, что в ООП стоит выделять еще одну немаловажную характеристику — абстракцию. Официально её не вносили в обязательные черты ООП, но списывать ее со счетов не стоит.

**Абстрагирование**— это способ выделить набор значимых характеристик объекта, исключая из рассмотрения не значимые  Соответственно, абстракция — это набор всех таких характеристик.

**Инкапсуляция**— это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя.

**Наследование**— это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом

**Полиморфизм**— это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Класс

class Имя\_Класса

Имя класса обязательно с большой буквы.

|  |
| --- |
| class Person  {    } |

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы.

Классы состоят из полей (переменные класса), свойств, методов.

Определим пару простых классов:

|  |
| --- |
| class Person  {  public int age;  public string name;  public void ShowInfo()  {  Console.WriteLine($"name {name}, {age} year old");  }  }  enum CarBody  {  sedan,  hachback,  crossover  }  enum Oil  {  gas,  gasoline,  diesel  }  enum Transmission  {  mechanical,  auto  }  class Car  {  public CarBody carType;  public Oil oilType;  public Transmission transmissionType;  public string brand;  public void ShowInfo()  {  Console.WriteLine($"Car brand {brand}, {carType}, transmition {transmissionType}");  }  } |

Тут мы определили два простых класса: Person и Car

В обоих классах определён метод ShowInfo() и несколько полей. У класса Car поля в основном состоят их перечислений.

Давайте определим несколько объектов класса Person в методе Main.

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Person per1;  per1 = new Person();  per1.age = 31;  per1.Name = "Павел";  Person per2 = new Person();  per2.age = 25;  per2.Name = "Костя";  per1.ShowInfo();  per2.ShowInfo();  Console.Read();  }  } |

Рассмотрим подробнее:

Так как классы являются ссылочным типом данных, его объявление и инициализация происходят через два этапа и с ключевым словом new.

Person per1 – мы объявляем новую переменную типа Person с именем per1, в данный момент переменная не ссылается ни на какой объект и имеет значение null, попытка обратиться к полям или методам закончится ошибкой NullReferenceException, которая означает, что вы попытались обратиться к несуществующему объекту.

Чтобы создать объект необходимо его проинициализировать:

per1 = new Person();

этой строчкой мы выделили память в компьютере под объект per1 класса Person. После этого мы можем обращаться к полям и методам класса.

Вывод программы, написанной выше:



В этой программе заметно, что мы создали два объекта, и оперировали данными, присущими только этим объектам.

Конструкторы

Если вызвать метод ShowInfo() до присвоения значения полям объекта per1, то мы увидим следующее:



Из этого можно сделать вывод: Поля класса инициализируются значениями по умолчанию, которые соответствуют их типу.

А что, если нам надо, чтобы основные поля класса были проинициализированны значениями уже при создания объекта?

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Конструктор имеет такое же имя как и класс и не имеет возвращаемого типа

|  |
| --- |
| class Person  {  public Person()// Конструктор.  {  }  } |

Отличие конструктора от остальных методов заключается в том, что он вызывается при создании объекта, и его основное назначение – инициализация класса и, если требуется передача классу начальных параметров, без которых класс не может правильно выполнять свою функцию.

Как и другие методы, конструктор может принимать аргументы, и на него распространяются те же правила, что и на обычные методы, за исключением возвращаемого типа.

Любой класс имеет конструктор «по умолчанию» ровно до тех пор пока мы не создадим другой конструктор. Конструктор по умолчанию это «пустой» конструктор, который не принимает аргументов, и не выполняет ни каких функций, по сути он выглядит также как в коде выше.

Мы можем создать собственные конструкторы:

|  |
| --- |
| class Person  {  public int age;  public string name;  public Person()  {  name = "Unknow";  age = 18;  }  public Person(string n)  {  name = n;  age = 18;  }  public Person(string n, int a)  {  name = n;  age = a;  }  public void ShowInfo()  {  Console.WriteLine($"name {name}, {age} year old");  }  } |

В этом классе мы определи три конструктора:

public Person()

public Person(string n)

public Person(string n, int a)

В первом конструкторе без параметров, мы присваиваем полям класса значения, которые мы установили, как по умолчанию.

Во втором конструкторе мы получаем имя и устанавливаем возраст.

В третьем примере мы получаем и имя и возраст.

Надо помнить, что при создании любого конструктора в классе, конструктор по умолчанию больше не существует. И если мы не определили конструктор, который не принимает аргументов, то класс нельзя будет создать без передачи необходимых параметров.

|  |
| --- |
| static void Main(string[] args)  {  Person per1 = new Person();  Person per2 = new Person("Костя");  Person per3 = new Person("Света", 23);  per1.ShowInfo(); // Unknow, 18  per2.ShowInfo(); // Костя, 18  per3.ShowInfo(); // Света, 23  Console.Read();  } |

Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

|  |
| --- |
| class Person  {  public string name;  public int age;  public Person() : this("Неизвестно")  {  }  public Person(string name) : this(name, 18)  {  }  public Person(string name, int age)  {  this.name = name;  this.age = age;  }  public void ShowInfo()  {  Console.WriteLine($"Name {name}, {age} years old");  }  } |

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

|  |
| --- |
| public Person(string name) : this(name, 18)  {  } |

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

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

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Инициализаторы объектов

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |
| --- |
| Person sasha = new Person { name = "Саша", age = 31 };  sasha.ShowInfo(); // Tom 31 |

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.

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.

Модификаоры доступа

Все члены класса - поля, методы, свойства - все они имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод.

В C# применяются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* **protected**: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* **private protected**: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal.

Все классы и структуры, определенные напрямую в пространствах имен и не являющиеся вложенными в другие классы, могут иметь только модификаторы public или internal.

|  |
| --- |
| public class State  {  int a; // все равно, что private int a;  private int b; // поле доступно только из текущего класса  protected int c; // доступно из текущего класса и производных классов  internal int d; // доступно в любом месте программы  protected internal int e; // доступно в любом месте программы и из классов-наследников  public int f; // доступно в любом месте программы, а также для других программ и сборок  protected private int g; // доступно из текущего класса и производных классов, которые определены в том же проекте  private void Display\_f()  {  Console.WriteLine($"Переменная f = {f}");  }  public void Display\_a()  {  Console.WriteLine($"Переменная a = {a}");  }  internal void Display\_b()  {  Console.WriteLine($"Переменная b = {b}");  }  protected void Display\_e()  {  Console.WriteLine($"Переменная e = {e}");  }  } |

Так как класс State объявлен с модификатором public, он будет доступен из любого места программы, а также из других программ и сборок. Класс State имеет пять полей для каждого уровня доступа. Плюс одна переменная без модификатора, которая является закрытой по умолчанию.

Также имеются четыре метода, которые будут выводить значения полей класса на экран. Обратите внимание, что так как все модификаторы позволяют использовать члены класса внутри данного класса, то и все переменные класса, в том числе закрытые, у нас доступны всем его методам, так как все находятся в контексте класса State.

Теперь посмотрим, как мы сможем использовать переменные нашего класса в программе (то есть в методе Main класса Program):

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  State state1 = new State();  // присвоить значение переменной a у нас не получится,  // так как она закрытая и класс Program ее не видит  // И данную строку среда подчеркнет как неправильную  state1.a = 4; //Ошибка, получить доступ нельзя  // то же самое относится и к переменной b  state1.b = 3; // Ошибка, получить доступ нельзя  // присвоить значение переменной с то же не получится,  // так как класс Program не является классом-наследником класса State  state1.c = 1; // Ошибка, получить доступ нельзя  // переменная d с модификатором internal доступна из любого места программы  // поэтому спокойно присваиваем ей значение  state1.d = 5;  // переменная e так же доступна из любого места программы  state1.e = 8;  // переменная f общедоступна  state1.f = 8;  // Попробуем вывести значения переменных  // Так как этот метод объявлен как private, мы можем использовать его только внутри класса State  state1.Display\_f(); // Ошибка, получить доступ нельзя  // Так как этот метод объявлен как protected, а класс Program не является наследником класса State  state1.Display\_e(); // Ошибка, получить доступ нельзя  // Общедоступный метод  state1.Display\_a();  // Метод доступен из любого места программы  state1.Display\_b();  Console.ReadLine();  }  } |

Таким образом, мы смогли установить только переменные d, e и f, так как их модификаторы позволяют использовать в данном контексте. И нам оказались доступны только два метода: state1.Display\_a() и state1.Display\_b(). Однако, так как значения переменных a и b не были установлены, то эти методы выведут нули, так как значение переменных типа int по умолчанию инициализируются нулями.

Несмотря на то, что модификаторы public и internal похожи по своему действию, но они имеют большое отличие. Классы и члены класса с модификатором public также будут доступны и другим программам, если данных класс поместить в динамическую библиотеку dll и потом ее использовать в этих программах.

Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы. Такое сокрытие называется **инкапсуляцией**.

Инкапсуляция

Инкапсуляция это Сокрытие данных от внешнего вмешательства.

Теория:

Представим на минутку, что мы оказались в конце позапрошлого века, когда Генри Форд ещё не придумал конвейер, а первые попытки создать автомобиль сталкивались с критикой властей по поводу того, что эти коптящие монстры загрязняют воздух и пугают лошадей. Представим, что для управления первым паровым автомобилем необходимо было знать, как устроен паровой котёл, постоянно подбрасывать уголь, следить за температурой, уровнем воды. При этом для поворота колёс использовать два рычага, каждый из которых поворачивает одно колесо в отдельности. Думаю, можно согласиться с тем, что вождение автомобиля того времени было весьма неудобным и трудным занятием.  
  
Теперь вернёмся в сегодняшний день к современным чудесам автопрома с коробкой-автоматом. На самом деле, по сути, ничего не изменилось. Бензонасос всё так же поставляет бензин в двигатель, дифференциалы обеспечивают поворот колёс на различающиеся углы, коленвал превращает поступательное движение поршня во вращательное движение колёс. Прогресс в другом. Сейчас все эти действия скрыты от пользователя и позволяют ему крутить руль и нажимать на педаль газа, не задумываясь, что в это время происходит с инжектором, дроссельной заслонкой и распредвалом. Именно сокрытие внутренних процессов, происходящих в автомобиле, позволяет эффективно его использовать даже тем, кто не является профессионалом-автомехаником с двадцатилетним стажем. Это сокрытие в ООП носит название инкапсуляции.  
  
**Инкапсуляция** – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.  
  
Инкапсуляция неразрывно связана с понятием интерфейса класса. По сути, всё то, что не входит в интерфейс, инкапсулируется в классе.

|  |
| --- |
| class Square  {  public double width;  public double height;  public double area;  public Square(double width, double height)  {  this.height = height;  this.width = width;  area = width \* height;  }  public void ShowInfo()  {  Console.WriteLine("Width {0}, height {1}, area {2}", width, height, area);  }  } |

Этот кода содержит ошибку!

Для предотвращения ошибок к нам на помощь приходят СВОЙСТВА совместно с модификаторами доступа

|  |
| --- |
| class Square  {  private double width;  private double height;  private double area;  public double Width  {  get { return width; }  set { width = value; }  }  public double Height  {  get { return height; }  set { height = value; }  }  public double Area  {  get { return area; }  set { area = value; }  }  public Square(double width, double height)  {  Height = height;  Width = width;  area = width \* height;  }  public void ShowInfo()  {  Console.WriteLine("Width {0}, height {1}, area {2}", width, height, area);  }  } |

Этот код ещё не решил ошибку, но уже скрыл поля класса, от внешнего изменения.

|  |
| --- |
| class Square  {  private double width;  private double height;  private double area;  public double Width  {  get { return width; }  set  {  width = value;  Area = width \* Height;  }  }  public double Height  {  get { return height; }  set  {  height = value;  Area = width \* Height;  }  }  public double Area  {  get { return area; }  private set { area = value; }  }  public Square(double width, double height)  {  Height = height;  Width = width;  area = width \* height;  }  public void ShowInfo()  {  Console.WriteLine("Width {0}, height {1}, area {2}", width, height, area);  }  } |

Данный класс полностью лишён той ошибки!

Внешне свойства ничем не отличаются от полей, но внутри самого класса они отличаются тем, что подобно методам содержат логику. Свойства определяются также как поля, но с добавлением блока **get/set**.

get и set — это **средства доступа к свойствам (accessors)**. getвыполняется при чтении свойства. Он должен возвращать значение того же типа, что и само свойство. set выполняется когда свойству присваивается значение. Оно имеет скрытый параметр value того же типа, что и само свойство.

Хотя доступ к полям и свойствам осуществляется одинаково, свойства в отличие от полей предоставляют полный контроль над процессом присвоения и получения значений (например, метод set может выбрасывать исключения и т.д.).

Если из средств доступа установлен только get, свойство будет доступно только для чтения (**read-only**), а если только set — только для записи (**write-only**). Обычно для свойства создается частное поле, в котором сохраняются данные, но это не обязательно: свойство может возвращать результаты расчета других данных.

Автоматические свойства

Часто бывает, что свойство не должно иметь логики присвоения или получения данных, а нужны только для разграничения доступа, и тогда создание обычных свойств создаёт много кода, и для решения этой проблемы существуют автоматические свойства:

Модификатор\_доступа тип имя { get; set; }

|  |
| --- |
| class Person  {  public string Name { get; set; }  public int Age { get; set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  } |

Так же можно разграничить права доступа для свойств:

|  |
| --- |
| class Person  {  public string Name { get; private set; }  public int Age { get; private set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  } |

При использовании автоматических свойств поля для свойств создаются компилятором, а не программистом, что существенно сокращает код, в будущем если понадобиться добавить логику для свойства, авто-свойство очень легко превратить в обычное.

Автосвойствам можно добавить значения по умолчанию, которые будут присвоены переменным сразу после создания объекта.

|  |
| --- |
| class Person  {  public string Name { get; private set; } = "Вася";  public int Age { get; private set; } = 18;  public Person(string name, int age)  {  Name = name;  Age = age;  }  } |

Структуры struct

Структуры схожи с классами, но имеют два ключевых отличия:

* структура — значимый тип, класс — ссылочный тип
* структуры не поддерживают наследования

Структуры могут содержать все члены доступные для класса, кроме конструктора без параметров.

Структура может быть использована вместо класса если симантика значимых типов более предпочтительны (например, в случае с числовыми типами). К тому же структуры более эффективны в плане производительности, т.к. на них расходуется меньше памяти.

У структур существует скрытый конструктор без параметров. Он используется по умолчанию и его нельзя переопределить. Он присваивает всем полям нулевые значения.

Структуре можно задать конструктор с параметрами, в нем необходимо явно присвоить значения всем полям.