

# **Лекция 7.**

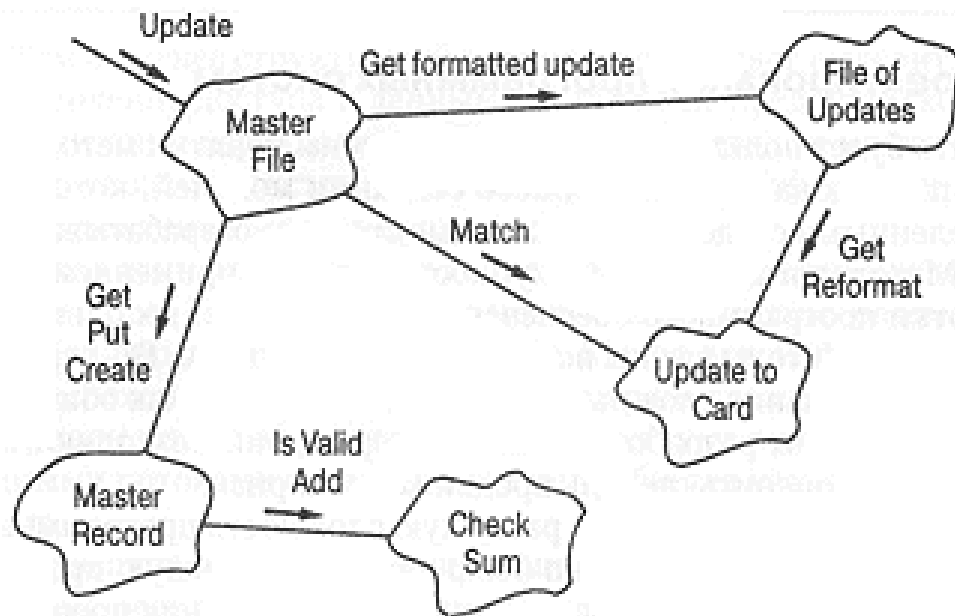
## **Основы ООП**

# Объектно-ориентированное программирование

- ООП - одна из самых распространенных «промышленных» парадигм программирования
- **Парадигма программирования** - это совокупность идей и понятий, определяющих стиль написания компьютерных программ
- Программы в ООП пишутся в терминах **объектов** и **классов**

# Объектно-ориентированное программирование

- С# является **объектно-ориентированным языком**
- Это означает что программа на С# представляет собой набор взаимодействующих **объектов**



# Объект

- **Объект** – это некоторая конкретная сущность (предмет, явление)
- **Примеры:**  
Стол, кошка, гроза, ноутбук, человек и т.д.
- Каждый объект обладает **состоянием, поведением и уникальностью**

# Состояние объекта

- **Состояние** – это набор характеристик объекта и их значений в данный момент времени
- В программировании состояние объектов задается при помощи переменных-полей
- **Пример для ноутбука:**
  - Высота, длина, ширина (вещественные числа)
  - Процент заряда (целое число)
  - Название модели (строка)
  - Заряжается сейчас или нет (**bool**) и т.д.
- Состояние может меняться под внешним воздействием, либо сам объект может менять свое состояние
- Например, со временем процент заряда падает



# Поведение объекта

- **Поведение** – это действия, которые может совершать объект и как объект может реагировать на воздействие со стороны других объектов
- Пример для ноутбука – его можно поставить на зарядку, и тогда процент будет расти
- Или выключатель – его можно включить или выключить
- В программировании поведение объекта задается при помощи функций-методов



# Уникальность объекта

- **Уникальность** объекта – это то, что отличает его от других объектов
- Например, каждый человек уникален
- Или есть два одинаковых стула, но это все равно два стула, а не один. То есть стулья уникальны – это 2 отдельных объекта
- В программировании уникальность задается расположением объекта в памяти компьютера



# Классы

- **Класс** – это вид **объектов** с одинаковой структурой и поведением
- Каждый объект обязательно **принадлежит** некоторому **классу**
- Если объект принадлежит некоторому классу, то говорят, что он является **экземпляром класса**
- То есть **объект** – это **экземпляр класса**



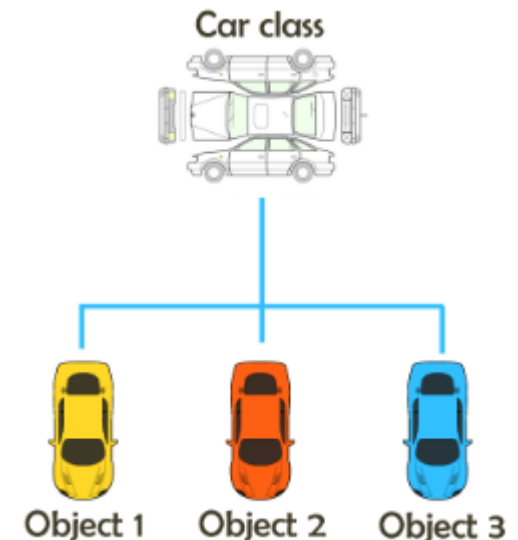
# Пример класса

- Пусть, у нас есть кошка Мурка
- Она является объектом класса Кошка
- Все кошки (то есть объекты класса Кошка) устроены и ведут себя схожим образом
- У всех них есть свое состояние – цвет, положение в пространстве, размеры и др.
- У них есть свое поведение – они могут ходить, бегать, реагировать на воздействия и т.д.



# Классы

- В программировании мы описываем **классы**
- В классе мы описываем, что в нем есть – какие поля, каких типов, какие есть методы, пишем их код
- А потом создаем сколько нам нужно **объектов (экземпляров)** этих классов и работаем с ними
- Т.е. класс – это как бы чертеж, описание, по которому потом можно создавать объекты, а объект – конкретная деталь, сделанная по этому чертежу

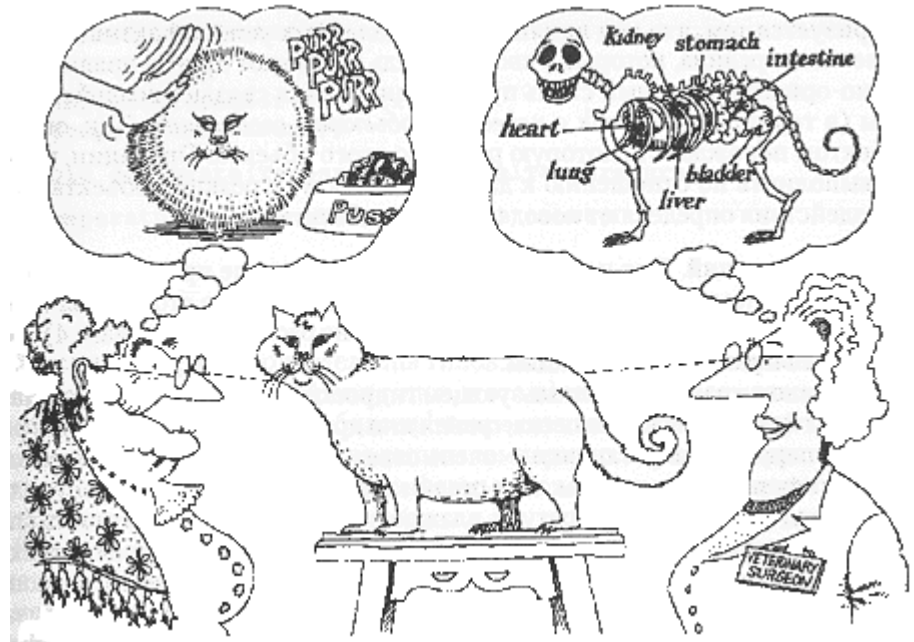


# Принципы ООП

- В ООП программы пишутся в терминах классов и объектов
- Принципы ООП:
  - Абстракция
  - Инкапсуляция
  - Наследование
  - Полиморфизм

# Абстракция

- **Абстракция** – выделение существенных характеристик объекта и существенного поведения, и отбрасывание несущественных характеристик и поведения
- Один и тот же объект реального мира для разных задач может быть представлен по-разному
- Важно выбирать абстракцию как можно более простую, но достаточную для задачи



# Пример абстракции 1

- Допустим, мы деканат и у нас такая задача – хранить список всех студентов
- Для этого выделим класс Студент
- Понятно, что студент является человеком, то есть у него есть пол, размеры, вес, возраст, у каждого человека много присущих ему черт
- Но для нашей задачи нам достаточно знать о студентах только ФИО, дату рождения, контактные данные, номер зачетки, номер группы

## Пример абстракции 2

- Допустим, мы военкомат и хотим хранить список всех студентов
- Для этого выделим класс Студент
- Для нашей задачи нам нужно знать ФИО, возраст, рост, состояние здоровья студента

## Пример абстракции 3

- Допустим, мы бухгалтерия университета и хотим хранить список студентов
- Для этого выделим класс Студент
- Для нашей задачи нам нужно знать ФИО, номер стипендиальной карты, категорию учащегося (например, получает стипендию или нет; бюджетник или нет)

# Абстракция

- Объект реального мира может быть одним и тем же
- Но в зависимости от задачи мы выбираем разные абстракции



# Абстракция

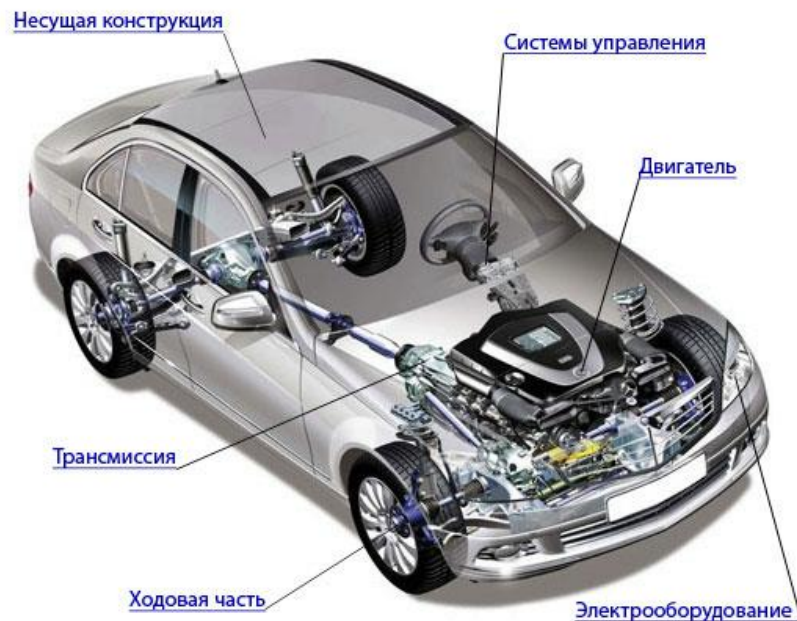
- Абстракция в основном охватывает внешние стороны объекта и не концентрируется на деталях реализации
- Пример абстракции – дверь
- В слове «дверь» не говорится из чего она сделана, ее размеры и так далее. Но мы уже понимаем что дверь можно открывать и закрывать. В этом и есть суть абстракции «дверь»

# Инкапсуляция

- **Инкапсуляция** – механизм языка, позволяющий объединить данные и методы, работающие с этими данными, в единый объект, и скрыть детали реализации от пользователя кода
  - Здесь видно 2 роли инкапсуляции – объединение в объект и сокрытие реализации
- В соответствии с принципом инкапсуляции, мы хотим максимально скрыть от **пользователя кода (т.е. программиста)** реализацию классов, а дать им только возможность работать с публичным интерфейсом
- Это позволяет легко подменять одну реализацию другой (можно спокойно менять то, что скрыто)

# Инкапсуляция

- Пример из жизни – устройство автомобиля
- Автомобиль состоит из огромного количества деталей, которые как-то друг с другом взаимодействуют
- Но чтобы водить автомобиль, не нужно знать многого – нужно только уметь работать с интерфейсом – руль, педали, коробка передач



# Инкапсуляция

- Так же и в коде – класс может быть очень сложно устроен, иметь вспомогательные функции и поля, но наружу предоставлять только функции, нужные другим
- И другим программистам даже не нужно знать как этот класс устроен внутри
- Надо только знать как этим классом пользоваться
- Пример – класс `Scanner` в Java или класс `StreamReader` в C#

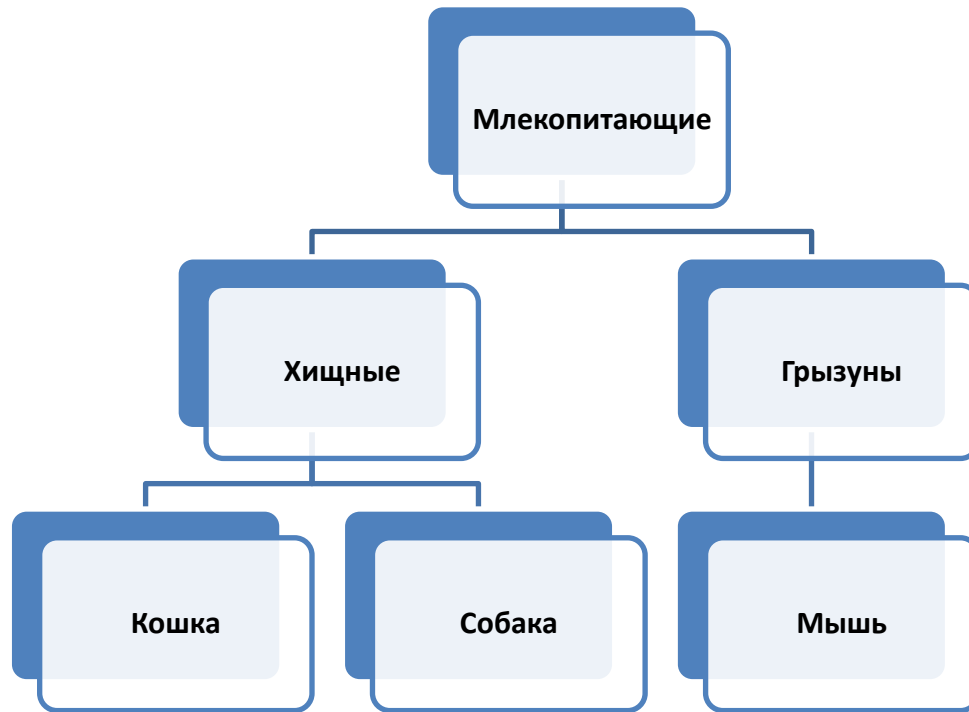
# Путаница - инкапсуляция и сокрытие

- Есть другое определение от Гради Буча:
- **Инкапсуляция** – процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение
  - В этом определении идет акцент только на **сокрытие реализации**
- Считается, что нужно различать **инкапсуляцию** и **сокрытие**
- Есть разные мнения что такое инкапсуляция – кто-то считает, что это только сокрытие, кто-то считает, что это только объединение данных и методов в класс, кто-то, что это и то и другое вместе
- Мы будем считать, что это и объединение и сокрытие

# Наследование

- Классы могут образовывать иерархию **наследования**
- Класс-наследник получает все свойства класса-родителя, может переопределять его черты, либо добавлять новые черты

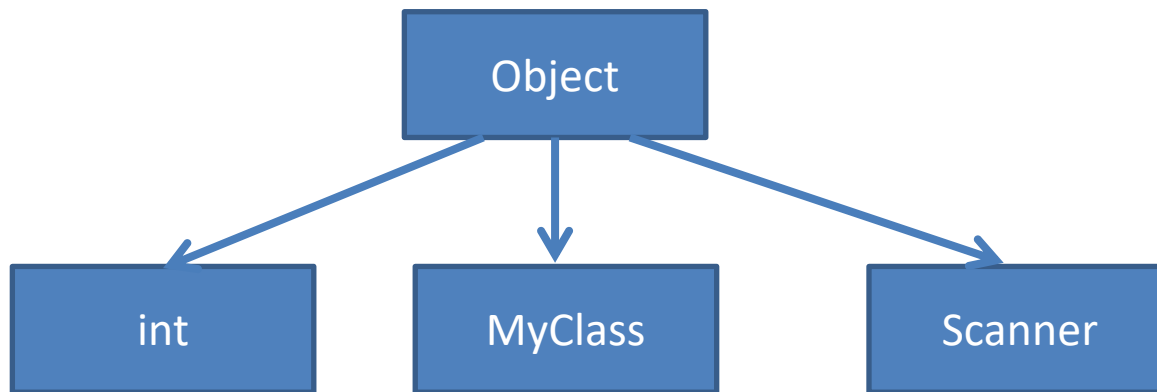
# Наследование



- Пример: биологическая классификация, рассмотрим кошку
- Так как кошка принадлежит классу млекопитающих, то она наследует свойства, присущие этому классу – кормление детей молоком
- Так как принадлежит классу хищников, то ест мясо и т.д.

# Типы данных в C#

- В C# абсолютно все типы являются классами
- И все типы наследуются от класса **Object**





# Подход без ООП со структурами

- Пусть у нас есть программа, в которой мы работаем с геометрическими точками на плоскости
- У нас есть сущность - точка с двумя координатами  $x$ ,  $y$
- В не ООП языках (например, в языке **C**) было такое понятие, как **структура**
- **Структура** – это тип данных, который внутри себя хранит несколько переменных
- `struct Point`  

```
{  
    double x;  
    double y;  
}
```
- Здесь и далее в презентации речь идет о структурах языка C, а не языка C#. В C# тоже есть понятие **структура**, но это понятие отличается

# Структуры

- `struct Point`  
`{`  
    `double x;`  
    `double y;`  
`}`
- Объявив такой тип, можно создавать его переменные и работать с ними
- `Point p = new Point();`  
`p.x = 4;`  
`p.y = 12;`
- В общем-то, похоже на класс, но у структур нет поведения. Они могут только хранить состояние

# Пример программы со структурами

- ```
public static void PrintPoint(Point p) {  
    Console.WriteLine("{0}, {1}", p.x, p.y);  
}
```
- ```
public static double GetDistance(Point p1, Point p2) {  
    return Math.Sqrt(Math.Pow(p1.x - p2.x, 2)  
        + Math.Pow(p1.y - p2.y, 2));  
}
```
- ```
public static void Main() {  
    Point p = new Point();  
    p.x = 3;  
    p.y = 5;  
    PrintPoint(p);  
}
```

# Краткий синтаксис создания структуры

- Есть еще такой синтаксис создания структуры:
- `Point p = { 3, 5 };`
- Мы указываем значения для полей через запятую по порядку
- Можно указывать не все поля

# Недостатки структур

- Полный доступ на чтение и запись ко всем полям структуры – любой код может читать и писать поля, что может приводить к ошибкам в коде
- Код функций, работающих с полями структуры, находится отдельно
- Небезопасная логика создания экземпляров структуры:
  - Нет возможности сделать валидацию входных данных
  - При использовании краткого синтаксиса заполнения полей структуры можно ошибиться – заполнить не все поля
    - Или в будущем изменится порядок полей, или добавятся новые поля, а мы забудем поменять код

# От структур к классам

- `class Point {`

- `private double x;`

- `private double y;`

В классе 2 переменных (поля)

- `public Point(double x, double y) {`

- `this.x = x;`

- `this.y = y;`

- `}`

Конструктор – функция  
инициализации нового объекта

- `public void Print() {`

- `Console.WriteLinef("{0}, {1}", x, y);`

- `}`

Функции могут обращаться  
к полям, и они не static

- `public double GetDistance(Point p) {`

- `return Math.Sqrt(Math.Pow(x - p.x, 2) + Math.Pow(y - p.y, 2));`

- `}`

- `}`

# От структур к классам

- ```
public static void Main() {  
    Point p = new Point(3, 5);  
    p.Print();  
}
```
- Сравним со структурами:
- ```
public static void Main() {  
    Point p = { 3, 5 };  
    PrintPoint(p);  
}
```
- ООП вариант проще для понимания, т.к. мы думаем в терминах объектов - мы просим точку распечататься
- А в варианте со структурами мы передаем точку в функцию, это более «машинный» подход

# Выгода использования классов

- Классы серьезно упрощают понимание кода
- Код легче модифицируется так как данные и функции находятся вместе – в коде класса
- Класс может скрывать то, чего другим знать не нужно (при помощи модификаторов видимости, например, `private`)
- Есть надежная логика инициализации новых объектов при помощи конструкторов



# Синтаксис класа

# Классы в C#

- `class Point`  
`{`  
    // члены класса: поля и методы  
`}`
- Каждый класс в C# может содержать поля (переменные), методы (функции), свойства (properties) и события
- Мы пока остановимся только на полях, методах и свойствах
- Поля определяют структуру класса, а методы – поведение класса

# Классы в С#

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void Print() {  
        Console.WriteLine("{0}, {1}", x, y);  
    }  
}
```

Имя класса

Поля (переменные)

Конструктор  
(специальная функция),  
вызываемая при создании  
объекта

Метод (функция)

# Порядок объявления членов класса

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void Print() {  
        Console.WriteLine("{0}, {1}", x, y);  
    }  
}
```
- Порядок членов класса неважен, но обычно поля пишут вверху, ниже пишут конструкторы, а ниже - методы

# Классы в C#

- class Point {  
 private double x;  
 private double y;

Если имя поля конфликтует с именем переменной, то обращаемся к нему через this

- public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
}

this – ключевое слово, обозначающего текущий объект (для которого вызвана функция)

- public void Print() {  
 Console.WriteLine("{0}, {1}", x, y);  
}

Можем всегда обращаться к полям и методам через this

# Конструкторы

- `class Main`  
`{`  
    `public static void Main(string[] args)`  
    `{`  
        `Point point = new Point(2, 4);`  
        `point.Print();`  
    `}`  
`}`
- **Конструктор** – специальная функция, которая позволяет создать и инициализировать экземпляр класса
- Конструктор нельзя вызвать явно, но он вызывается если создавать объект при помощи оператора **new**

# Конструкторы

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```
- При объявлении функции-конструктора не указывается возвращаемый тип. Конструктор ничего не возвращает
- Имя конструктора всегда совпадает с именем класса

# Конструкторы

- Класс может иметь несколько конструкторов
- Это будет перегрузка, как для обычных методов

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point() {  
    }  
}
```



# Вызов конструкторов друг из друга

- Чтобы не дублировать код конструкторов, в конструкторе есть возможность вызывать другие конструкторы через ключевое слово `this`

- ```
public class Point {  
    private double x;  
    private double y;
```

```
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }
```

```
    public Point() : this(0, 0) {  
    }  
}
```

В теле конструктора может идти код, он выполнится после вызова другого конструктора

# Конструктор по умолчанию

- Если при объявлении класса не создавать конструктор, то компилятор C# сам генерирует **конструктор по умолчанию** (он без аргументов), который ничего не делает, а только вызывает конструктор класса-родителя
- Если в классе создать любой конструктор с аргументами, то компилятор не создает конструктор по умолчанию
- В этом случае если понадобится создать конструктор без аргументов, то его нужно будет объявить самим


# Обращение к полям и методам классов

- ```
class Main
{
    public static void Main(string[] args)
    {
        Point point = new Point(3, 2);
        point.Print();
    }
}
```
- Обращение к полям и методам объекта осуществляется через оператор точка
- Для членов класса могут иметься разные **права доступа**. Они задаются при объявлении класса при помощи **модификаторов видимости**, например, `public` и `private`. Еще есть `protected` и package видимость
- Если прав недостаточно, то обращение к члену класса приведет к ошибке компиляции

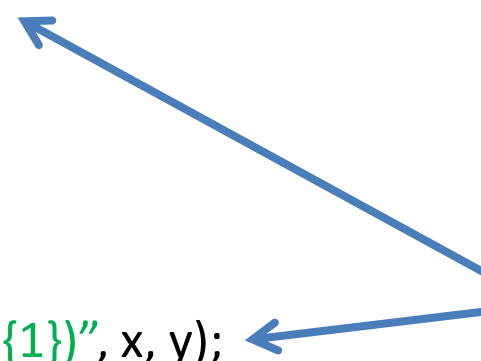
# Классы в С#

- class Point {  
 private double x;  
 private double y;  
  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void Print() {  
 Console.WriteLine("{0}, {1}", x, y);  
 }  
}

private члены класса  
видны только  
функциям внутри  
класса



public члены класса  
видны всем



# Обращение к полям и методам классов

- Модификаторы видимости и есть средство инкапсуляции в С# – они позволяют скрыть реализацию класса, а наружу выставлять только то, что должны использовать пользователи класса
- **Поля всегда должны быть `private`!! Если к ним все же нужен доступ, то для этого должны использоваться методы**

# Обращение к полям и методам классов

- Поля всегда должны быть **private**!! Если к ним все же нужен доступ, то для этого должны использоваться **методы**

- ```
class Point {  
    private double x;  
  
    public double GetX() {  
        return x;  
    }  
  
    public void SetX(double x) {  
        this.x = x;  
    }  
}
```

Соглашение именования –  
методы для получения  
значений должны начинаться с  
get, а для установки значения – с  
set

Методы get называют  
геттерами, методы set -  
сеттерами

Не обязательно иметь  
оба

# Зачем поля private?

- **Достоинства:**

- Пользователи кода теперь не могут вмешиваться во внутренние дела класса, например, присвоить полю недопустимое значение
- Если имя поля изменится, или поле вообще исчезнет, то метод можно оставить с прежним именем, и тогда это изменение не затронет код, который использовал этот метод
- Метод может выполнять дополнительную работу: проверять корректность данных, сохранять сообщения в лог и т.д.

- **Недостатки:**

- Некоторое падение производительности т.к. получить значение поля дешевле, чем вызвать метод. Но производительность часто не важна

# Свойства в С#

- В С# вместо геттера и сеттера используют **свойства (properties)**

- ```
class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Свойство выглядит для пользователя как поле, но по факту – это 2 метода – геттер и сеттер

Не обязательно иметь get и set одновременно, можно делать и только один из них

В set ключевое слово value обозначает передаваемое значение



# Свойства в С#

- В С# вместо геттера и сеттера используют **свойства (properties)**
- ```
public static void Main()  
{  
    Person p = new Person();  
    p.Name = "Ivan";  
    Console.WriteLine(p.Name);  
}
```
- В С# свойства – более предпочтительный вариант, чем геттеры и сеттеры

Свойство выглядит для пользователя как поле, но по факту – это 2 метода – геттер и сеттер

# Автоматические свойства

- Для свойств есть краткий синтаксис
- ```
class Person  
{  
    public string Name { get; set; }  
}
```
- Заметим, что тут не нужно писать поле и код в `get`, `set`
- Для таких свойств автоматически генерируется приватное поле, и свойство работает с ним
- Этот вариант более удобен, рекомендуется использовать его

# Нестатические члены класса

- class Person  
{  
 private string name;  
  
 public Person(string name)  
 {  
 this.name = name;  
 }  
  
 public string GetName()  
 {  
 return this.name;  
 }  
}

Поле name будет свое у каждого экземпляра класса Person

Методы класса могут работать с полями объекта. На сам объект можно сослаться при помощи слова this

# Статические члены

Статические члены относятся не к конкретным экземплярам, а к классу в целом

- class Person
  - {
    - private string name;
    - public static readonly int MaxNameLength = 100;
    - public Person(string name)
      - {
      - this.name = name;
      - }
    - public string GetName()
      - {
      - return name;
      - }
    - public static string FormatName()
      - {
      - // код
      - }

Статические поля существуют в единственном экземпляре

Чтобы работать со статическими членами не нужно создавать объекты класса

# Статические члены класса

- ```
class Person
{
    public static readonly int MaxNameLength = 100;
    public static string FormatName(string name)
    {
        // возвращает имя с инициалами
    }
}
```
- Как обращаться к статическим методам и полям:
- ```
public static void Main()
{
    int maxLength = Person.MaxNameLength;
    string formattedName = Person.FormatName("Ivan Ivanov");
}
```

# Статические члены класса

- Мы уже много работали со статическими методами и полями
- Например, мы использовали класс `Math` и его статические члены:
  - `Math.PI` – статическое поле-константа
  - `Math.Abs(x)` – получение модуля числа и т.д.
- Такие классы, как `Math`, которые содержат только статические методы и статические константы, называются **классами-утилитами**

# Static и не-static

|                | He static                             | Static                                                                            |
|----------------|---------------------------------------|-----------------------------------------------------------------------------------|
| <b>В целом</b> | Относится к объекту                   | Относится к классу в целом                                                        |
| <b>Поля</b>    | Это поле будет у каждого объекта свое | Поле будет одно на весь класс. Оно хранится не в объектах, а отдельно в программе |
| <b>Методы</b>  | Метод вызывается только от объекта    | Метод вызывается от класса в целом                                                |

# Задача

- Объявить класс **Contact** для хранения фамилии, имени и номера телефона человека
- Для каждого поля объявить геттер и сеттер
- В функции Main объявить несколько переменных класса **Contact** и поработать с ними – повызывать геттеры и сеттеры



# Задача на дом «Range»

- Создать класс `Range` (непрерывный вещественный числовой диапазон на прямой). В нём:
  1. Объявить два вещественных поля `from`, `to`
  2. Описать конструктор, при помощи которого заполняются поля
  3. Реализовать геттеры и сеттеры для полей
  4. Сделать метод для получения длины диапазона
  5. Сделать метод `IsInside`, который принимает вещественное число и возвращает `bool` – результат проверки, принадлежит ли число диапазону
- После этого написать небольшую программу с использованием этого класса

# Структура программ

# Файлы и типы данных

- Программы на C# обычно состоят из многих файлов
- В C# есть много разных категорий **типов данных** – классы, структуры, интерфейсы и др.
- В каждом файле может находиться один или более типов, но принято выносить каждый тип в отдельный файл
- Дальше мы будем рассматривать классы, но это относится и к другим типам

# Структура программ

- Для структурирования кода в C# есть 3 отдельные вещи:
  - Папки
    - Отвечают за структурирование файлов на диске
  - Пространства имен (**namespaces**)
    - Отвечают за логическое структурирование типов и за разрешение конфликтов имен между типами
  - Сборки (**assemblies**)
    - Отвечают за доступ к типам и за физическое разделение типов по разным файлам после компиляции

# Папки

- Под папками здесь имеются в виду обычные папки в файловой системе
- Папки используются для структурирования файлов в проекте и на диске
- Их можно создавать через Visual Studio или просто на диске внутри проекта
- Папки не влияют на доступ к типам и на имена типов

# Пространства имен

- **Пространство имен (namespace)** – это логическая группа типов
- В `namespace` стараются помещать близкие друг к другу типы
- Например, в один `namespace` можно поместить классы GUI – графического интерфейса, а в другой – классы логики программы

# Пространства имен

- Указать к какому пространству имен относится тип, можно при помощи блока **namespace**
- **Файл Program.cs:**
- `namespace` Academits.Courses

```
{  
    public class Program  
    {  
        public static void Main()  
        {  
            //...  
        }  
    }  
}
```

Класс Program лежит в  
пространстве имен  
Academits.Courses

# Пространства имен

- В новых версиях C# `namespace` можно указать так:
- **Файл Program.cs:**
- `namespace` Academits.Courses;

```
public class Program
{
    public static void Main()
    {
        //...
    }
}
```

Все типы из этого файла  
будут лежать в  
пространстве имен  
Academits.Courses



# Пространства имен

- `namespace` Academits.Courses  
 {  
 `// классы`  
 }
- Пространства имен могут вкладываться друг в друга.  
 Запись **Academits.Courses** означает что есть `namespace` **Academits**, а в нем есть вложенный `namespace` **Courses**
- Пространства имен никак не влияют на структуру папок проекта. Более того, в разных проектах (**project**'ах) можно иметь одно и то же пространство имен

# Зачем нужны пространства имен?

- Пространства имен позволяют:
  - логически структурировать типы
  - избежать конфликтов имен. Благодаря пространствам имен можно давать разным типам одинаковые имена, если эти типы лежат в разных пространствах имен
- В C# именем типа является не просто имя, которое мы указываем при объявлении типа, а имя пространства имен + имя типа
- Наш класс `Program` на самом деле называется `Academits.Courses.Program`

# Имена пространств имен

- Имена `namespace`'ов следует делать уникальными для всего мира, чтобы никогда не возникало конфликтов имен с чужим кодом
- Поэтому для уникальности, обычно, компании используют свое название
- Например, компания у нас Academ IT School (можно сократить до Academits)
- Поэтому `namespace` будет: Academits
- Для каждого проекта делается свой `namespace`. Например, для проекта Virtual Manager: Academits.VirtualManager

# Какие имена давать?

- Для студентов можно порекомендовать что-то такое:
  - `Academits.Ivanov`

# namespace'ы стандартной библиотеки

- Стандартная библиотека C# также разделена на множество пространств имен
- Например:
  - Основные классы находятся в пространстве имен **System**, например, класс **object**
  - Классы для работы со вводом и выводом находятся в пакете **System.IO**  
Например, это классы потоков ввода и вывода, файлы и т.д.
  - И т.д.

# using

- Любой тип может использовать любые типы из других пространств имен (только если они не **internal** и при этом не находятся в другой сборке)
- Но обращаться к типам других пространств имен можно только по **квалифицированному** (полному имени) – имя пространства имен + имя типа
- Например, если хотим использовать класс **System.Console**, то придется писать: **System.Console.WriteLine(“Текст”);**
- Чтобы все время не писать полные имена типов, а использовать только имя типа, существует инструкция **using**
- Visual Studio по умолчанию вставляет несколько частых **using**’ов в начало новых файлов, чтобы их не писать

# Неявные using'и

- В .NET 6 появились неявные using'и
- Например, для консольных проектов во всех файлах неявно добавлены следующие using'и:
  - `using System;`  
`using System.IO;`  
`using System.Collections.Generic;`  
`using System.Linq;`  
`using System.Net.Http;`  
`using System.Threading;`  
`using System.Threading.Tasks;`
- Поэтому их можно не писать
- При желании эту фичу можно отключить в настройках **project'a**

# using пространства имен

```
using System;
```

```
public class Main
{
    public static void Main()
    {
        Console.WriteLine("123");
    }
}
```

Теперь если в этом файле встретится имя типа, который есть в System, то компилятор будет считать, что мы используем этот тип



# Разрешение неоднозначности

```
using System;  
using Academits;  
  
using MyConsole = Academits.Console;  
  
public class Main  
{  
    public static void Main()  
    {  
        System.Console.WriteLine("Обычная консоль");  
        MyConsole.WriteLine("Своя консоль");  
    }  
}
```

Если нужно использовать оба типа, то одному из них можно дать псевдоним (любой) и пользоваться им

Либо всегда можно обратиться по полному имени типа

# Соглашения про namespace'ы и папки

- Если файл находится непосредственно внутри **project'**а, то в этом файле имя **namespace'**а должно совпадать с именем **project'**а
- Если файл вложен в некоторые папки внутри **project'**а, то эти папки нужно добавлять в **namespace** в этом файле
  - Каждая папка – отдельный уровень вложенности **namespace'**а
- Например, пусть **project** называется **Project**, класс называется **Test**, и он находится внутри папки **Folder2**, которая находится внутри папки **Folder1** в **project'**е
- Тогда **namespace** должен быть: **Project.Folder1.Folder2**
- Visual Studio использует эти соглашения при создании новых файлов с кодом

# Сборки

- **Сборка (assembly)** – это единица развертывания в .NET
- Каждый **project** в Visual Studio компилируется в отдельную **сборку**
- Сборка представляет из себя файл **.exe** или **.dll**
- Сборка содержит типы и ресурсы

# Доступ к типам

- В .NET можно сослаться из одной сборки на другую (в **Visual Studio** через раздел **Dependencies**), и тогда можно использовать типы этой сборки, если к ним есть доступ
- Если тип объявлен как **public**, то к нему можно обращаться из других сборок, если сослаться на эту сборку
- Если тип объявлен как **internal** (или без модификатора), то доступ к этому типу есть только в его сборке

# Чтение на дом

- Рекомендую дома читать эти курсы:
- <http://metanit.com/sharp/tutorial/>
- [http://professorweb.ru/my/csharp/charp\\_theory/level1/infocs\\_harp.php](http://professorweb.ru/my/csharp/charp_theory/level1/infocs_harp.php)
- И любые другие материалы, какие хочется
- В свободное время читайте эти курсы, задавайте вопросы