

ЛК 6 – ЛК 7. Классы и структуры

Основы ООП

- ООП - методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром конкретного класса.

Парадигмы ООП

- **Абстракция**
- **Инкапсуляция**
- **Полиморфизм**
- **Наслédование**

Абстракция

- Абстракция - это выделение общих характеристик объекта, исключая набор незначительных.
- С помощью принципа абстракции данных, данные преобразуются в объекты. Данные обрабатываются в виде цепочки сообщений между отдельными объектами. Все объекты проявляют свои уникальные признаки поведения. Огромный плюс абстракции в том, что она отделяет реализацию объектов от их деталей, что в свою очередь позволяет управлять функциями высокого уровня через функции низкого уровня.

Инкапсуляция

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

Полиморфизм

- Полиморфизм (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных.
- В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Наслédование

- **Наслédование** — механизм ООП, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

Классы

ПОНЯТИЕ КЛАССА. ОПИСАНИЕ КЛАССА. ЧЛЕНЫ КЛАССА.

Определение класса в ООП

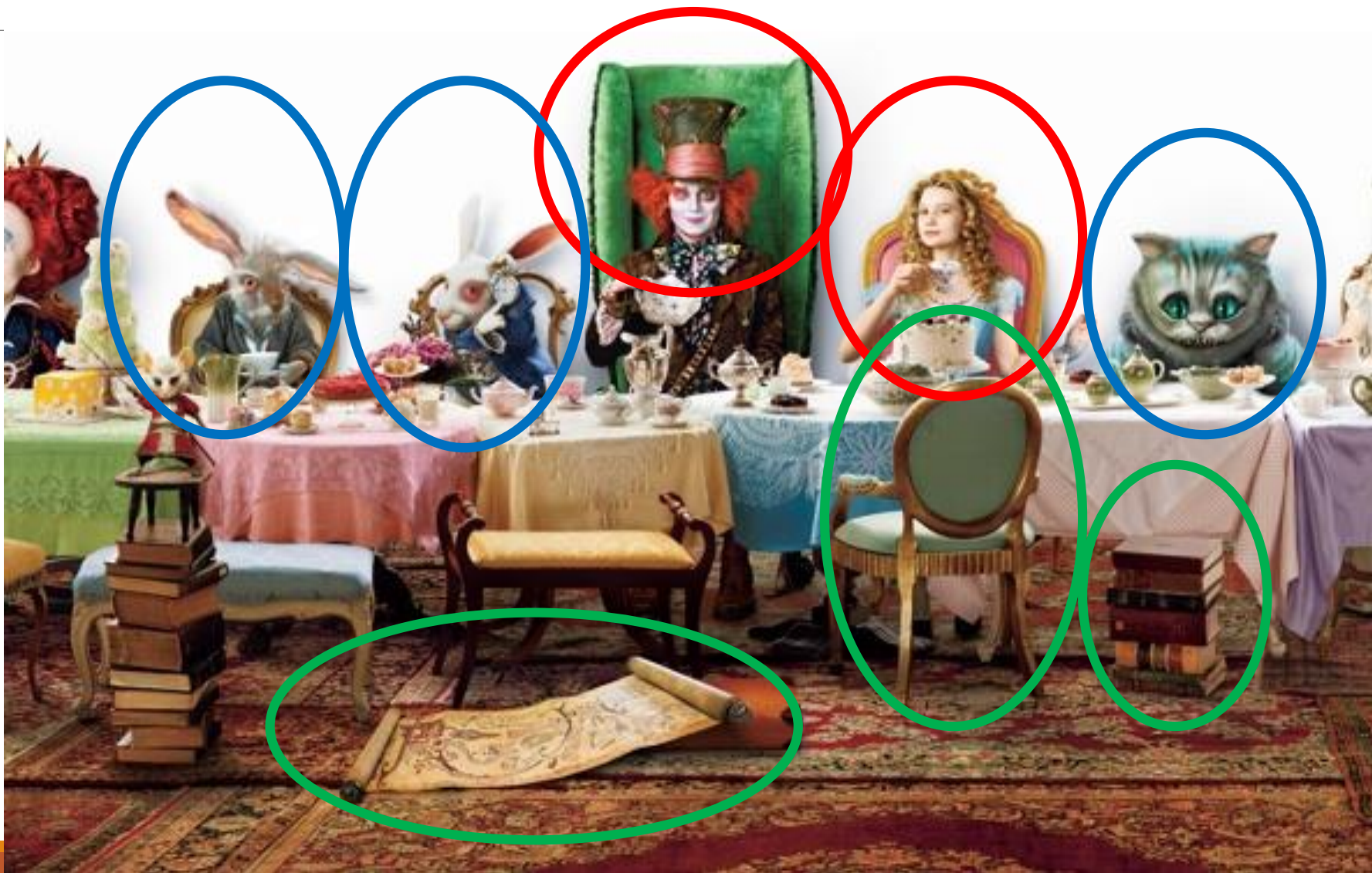
- Класс – это тип данных, задающий реализацию некоторой абстракции данных (сущности), характерной для моделируемой предметной области.

Проектирование в ООП

Объектно-ориентированная разработка программной системы основана на стиле, называемом проектированием от данных. Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи.

Каждая из таких абстракций реализуется в виде класса, которые и становятся модулями - архитектурными единицами построения нашей системы. В основе класса лежит абстрактный тип данных.





Шляпник

Мартовский заяц

Кролик

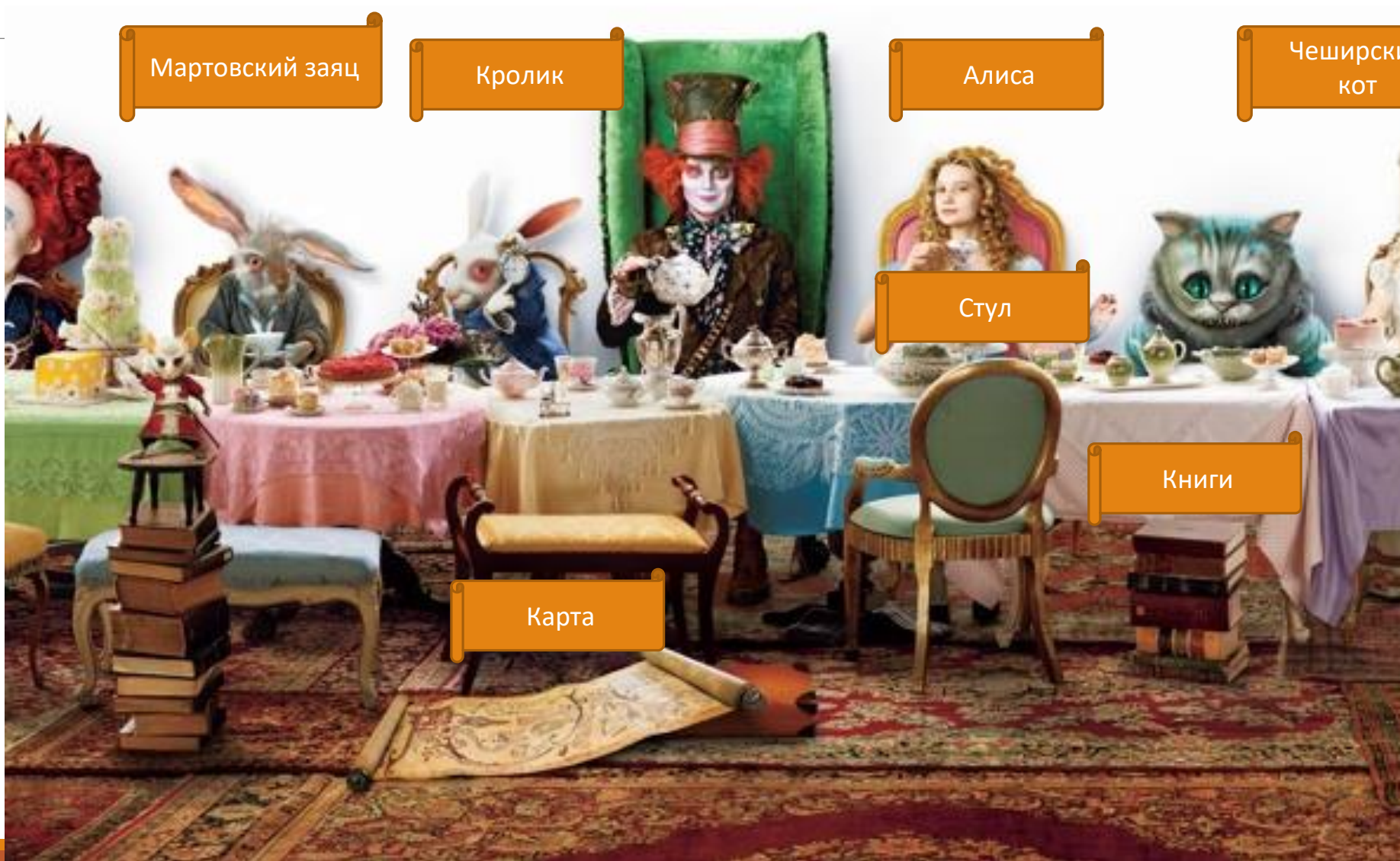
Алиса

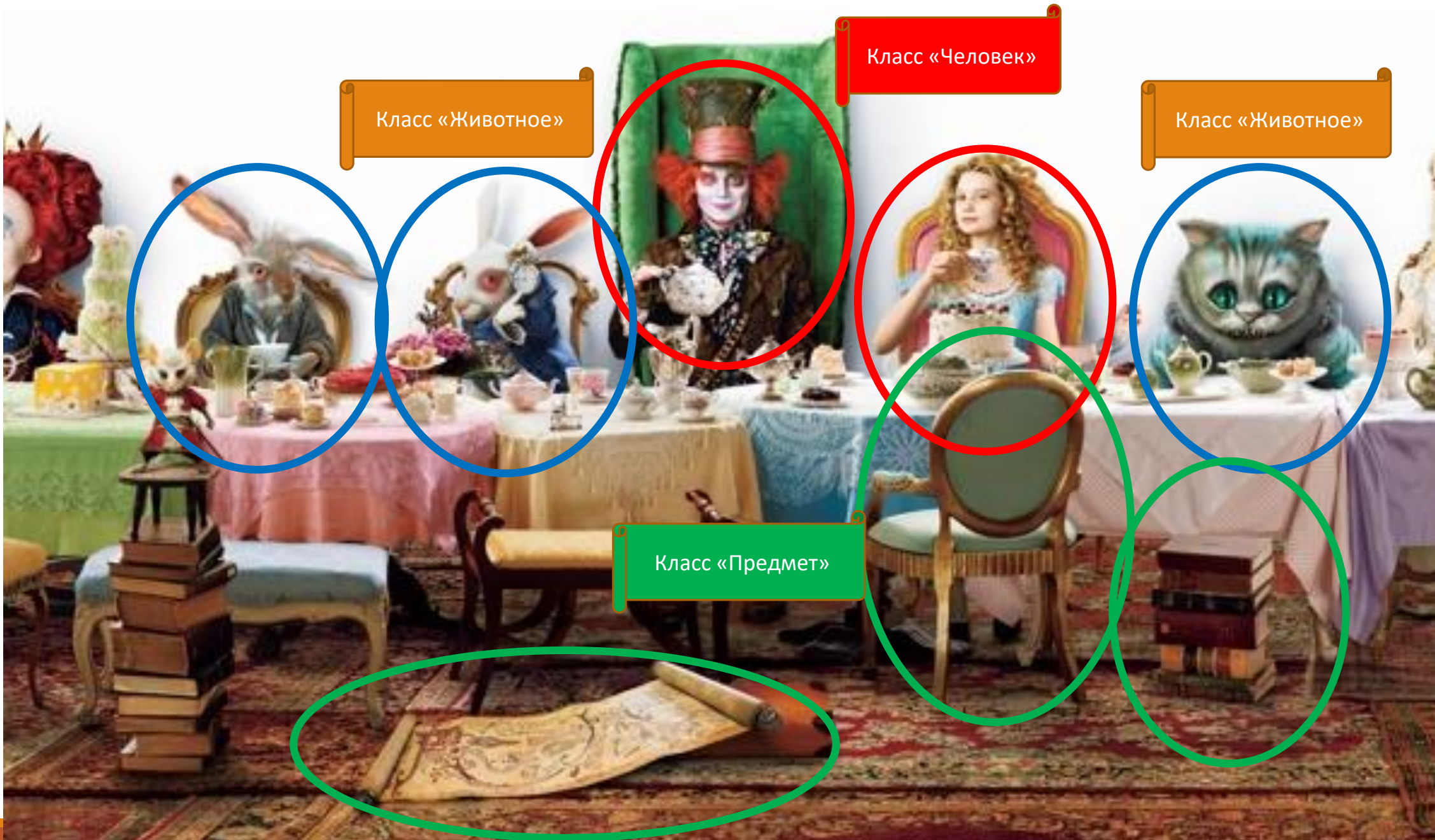
Чеширский кот

Стул

Книги

Карта





Класс «Животное»

Класс «Человек»

Класс «Животное»

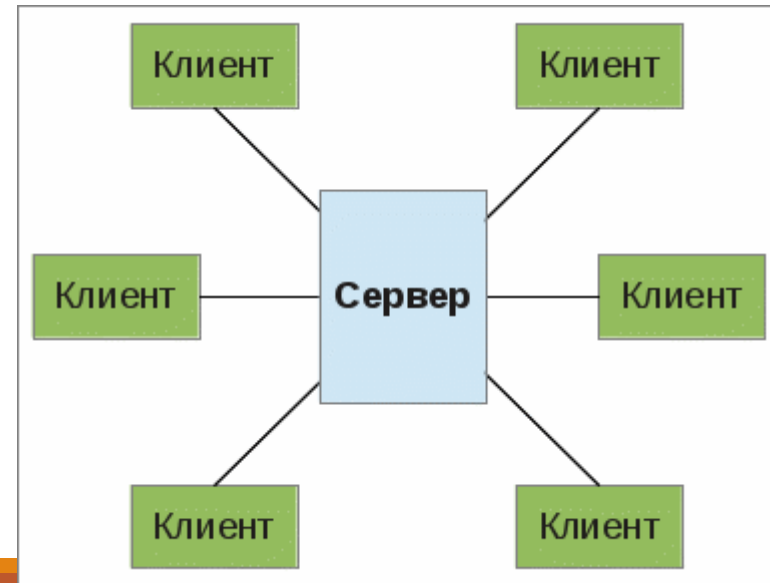
Класс «Предмет»

Проектирование в ООП

- Некоторые сущности могут включать в себя другие:
 - группа состоит из студентов
 - окна являются частью стены
 - светильники являются частью стены или потолка

Проектирование в ООП

- В ООП используются понятия **клиент** и **сервер**. Сервер – тот, кто предоставляет услугу (в нашем случае – класс), клиент – тот, кто использует этот класс (программист, возможно тот же, кто писал класс, возможно – другой).
- В процессе обучения вашим клиентом будет класс Program с методом `main()`.



Общая форма определения класса

```
class имя_класса
{
    // Объявление полей
    доступ тип переменная1;
    доступ тип переменная2;
    // . . .
    доступ тип переменнаяN;
    // Объявление методов
    доступ тип метод1(параметры)
    {
        //тело метода
    }
    доступ тип метод2(параметры)
    {
        // тело метода
    }
    доступ тип методM(параметры)
    {
        // тело метода
    }
}
```

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 иборок (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.

Класс

- Код и данные, которые составляют класс, называются членами класса. Данные, определенные в классе, называются полями класса, а код, который оперирует этими данными, — методами.
- Поля - это переменные класса
- Метод — это функция.
- Объекты - это экземпляры класса

Объявление класса

```
class Human  
{ }
```

```
class Animal  
{ }
```

```
class Thing  
{ }
```

Объект

- Объект - это конкретный экземпляр класса.
- при создании объекта выделяется память для хранения его переменных (полей) и поля инициализируются данными
- Каждый объект обладает своими полями
- Методы общие для всего класса

Шляпник и Алиса – экземпляры класса «Человек»

Создание объектов класса

```
Human Hatter = new Human();  
Human Alice = new Human();  
Animal MarchHare = new Animal();  
Animal CheshireCat = new Animal();
```

Создание объекта

Имя объекта

Имя класса

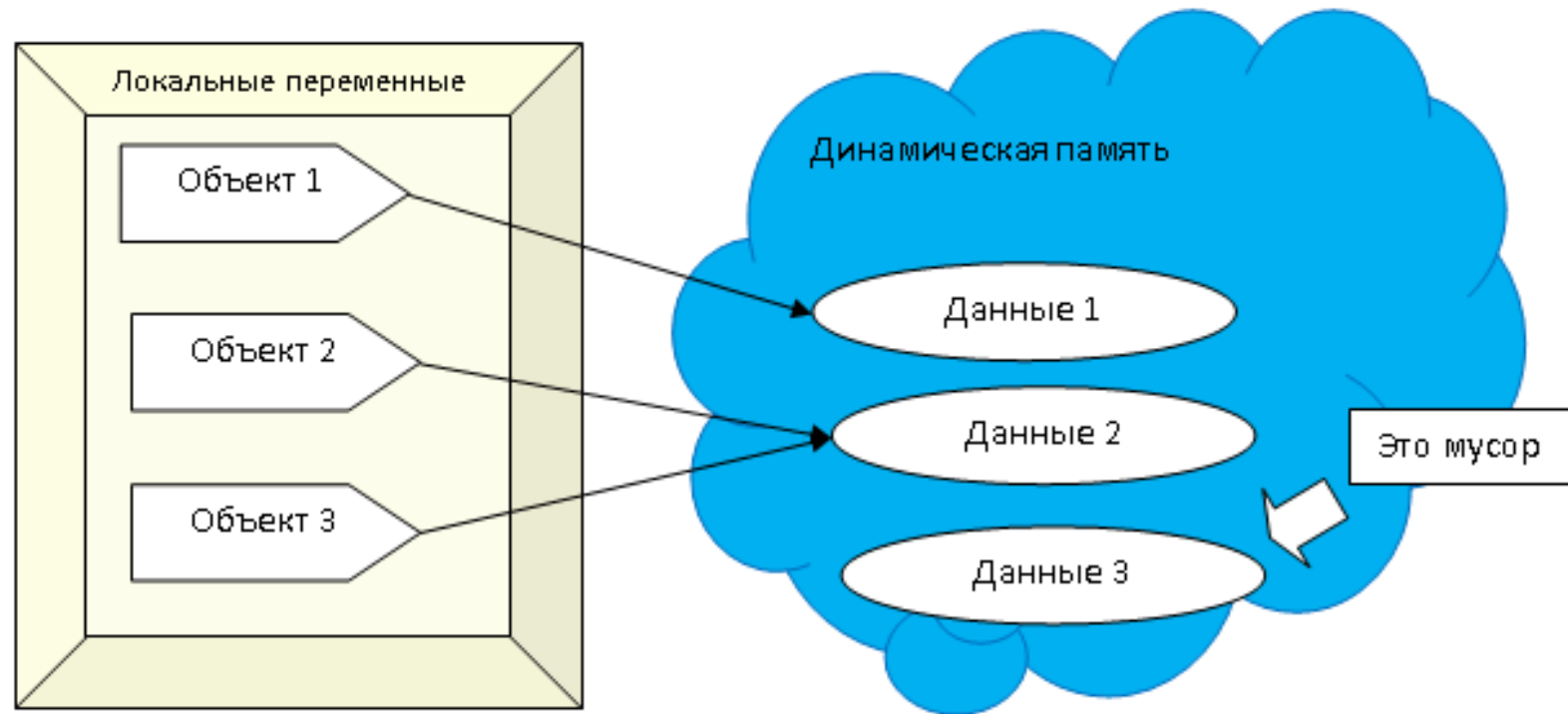
Создание объектов класса

```
Animal WhiteRabbit;
```

```
WhiteRabbit = new Animal();
```

Объекты класса являются данными **ССЫЛОЧНОГО** типа

Объект – ссылочный тип данных



Свойства объектов

Свойства автомобиля:

- Марка (текст)
- Мощность двигателя (число)
- Максимальная скорость (число)
- В наличии (логический тип)



Поля класса

Свойства объектов, описанные в классе, называются полями:

```
class Car
{
    public string Model;
    private int EnginePower;
    public int MaxSpeed;
    bool IsAvailable;
}
```

Доступ к полям класса

```
Car MyCar = new Car();
```

```
MyCar.
```

- ⊗ Equals
- ⊗ GetHashCode
- ⊗ GetType
- MaxSpeed
- Model
- ⊗ ToString

```
string Car.Model
```

Доступ к полям класса

```
Car MyCar = new Car();  
MyCar.Model = "Volvo";  
MyCar.MaxSpeed = 180;
```

Действия, выполняемые объектами

- Начать движение
- Остановиться
- Увеличить скорость
- Уменьшить скорость
- Показать текущую скорость



Методы класса

- Методы — это функции (процедуры или подпрограммы), которые манипулируют данными, определенными в классе, и во многих случаях обеспечивают доступ к этим данным.

Методы класса

доступ тип_возврата имя (список_параметров)

{

// тело метода

}

private (по умолчанию)
public

void (если ничего не
возвращает)

может быть пустым

Методы класса

```
public void Start()  
{  
    speed = 20;  
}
```

```
public void Stop()  
{  
    speed = 0;  
}
```

Возвращение значений методами

```
public int SpeedUp()  
{  
    speed += 20;  
    // Если полученная скорость больше  
максимальной  
    if (speed > MaxSpeed)  
    {  
        speed = MaxSpeed;  
    }  
    return speed;  
}
```

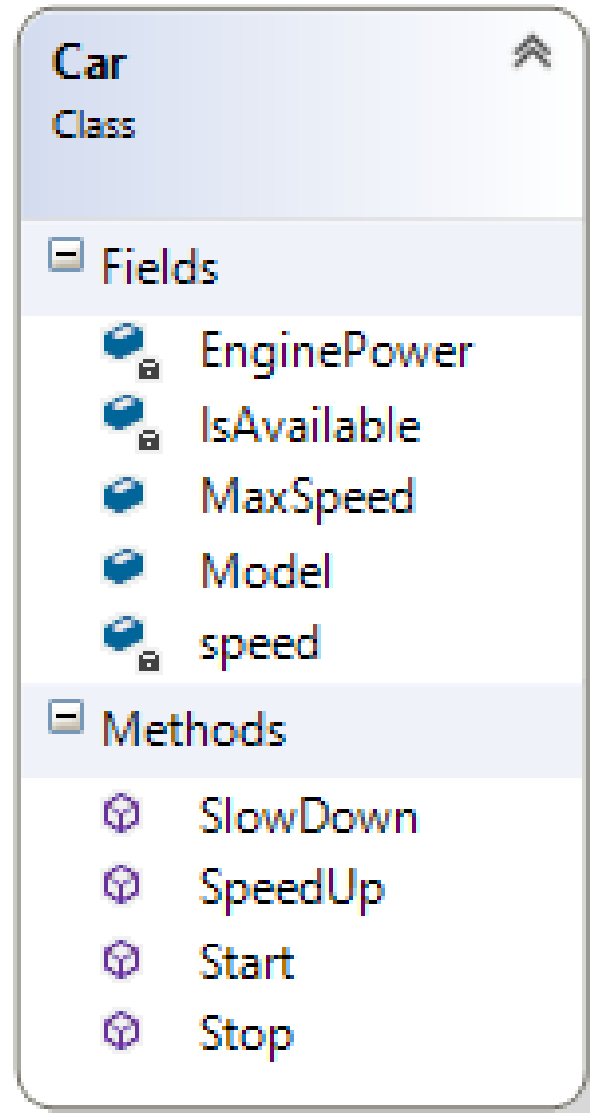
Задача

Написать метод `SlowDown`, уменьшающий скорость на 20 км/ч

Возвращение значений методами

```
public int SlowDown()  
{  
    if (speed > 20)  
    {  
        speed -= 20;  
    }  
    else  
    {  
        speed = 0;  
    }  
    return speed;  
}
```

Члены класса на UML-диаграмме



Использование методов класса

Изменим методы

```
public void Start()  
{  
    if (speed == 0) speed = 20;  
    Console.WriteLine("Машина поехала");  
}
```

```
public void Stop()  
{  
    speed = 0;  
    Console.WriteLine("Машина  
остановилась");  
}
```

Изменим методы

```
public int SpeedUp()
{
    speed += 20;
    if (speed > MaxSpeed) // Если полученная скорость больше максимальной
    {
        speed = MaxSpeed;
        Console.WriteLine("Достигнута максимальная скорость");
    }
    Console.WriteLine("Скорость=" + speed);
    return speed;
}

public int SlowDown()
{
    if (speed > 20)
    {
        speed -= 20;
        Console.WriteLine("Скорость=" + speed);
    }
    else
    {
        speed = 0;
        Console.WriteLine("Машина остановилась");
    }
    return speed;
}
```


Применим методы:

```
static void Main(string[] args)
{
    Car MyCar = new Car();
    MyCar.Model = "Volvo";
    MyCar.MaxSpeed = 40;
    MyCar.SpeedUp();
    MyCar.Start();
    MyCar.SpeedUp();
    MyCar.SpeedUp();
    MyCar.Stop();
}
```

Результат

C:\WINDOWS\system32\cmd.exe

Скорость=0

Машина поехала

Скорость=40

Скорость=40

Машина остановилась

Для продолжения нажмите любую клавишу . . .

Задача

Сделать так, чтобы при вызове метода Stop машина останавливалась плавно, с шагом 20 км/ч

```
public bool SlowDown()  
{  
    if (speed > 20)  
    {  
        speed -= 20;  
        Console.WriteLine("Скорость=" + speed);  
        return true;  
    }  
    else  
    {  
        speed = 0;  
        Console.WriteLine("Машина остановилась");  
        return false;  
    }  
}
```

```
public void Stop()  
{  
    while (SlowDown())  
    { }  
}
```

Передача параметров методу

Задача: Создать метод, позволяющий сразу установить нужную скорость.

```
public void SetSpeed(int speed)
{
    if (this.speed==0)
    {
        Start();
    }
    this.speed = speed;
    Console.WriteLine("Скорость="
speed);
}
```

+

```
static void Main(string[] args)
{
    Car MyCar = new Car();
    MyCar.Model = "Volvo";
    MyCar.MaxSpeed = 120;
    MyCar.SetSpeed(80);
    MyCar.Stop();
}
```

Параметры методов

- *Список формальных параметров метода* – это набор элементов, разделенных запятыми. Каждый элемент списка имеет следующий формат:
- [<модификатор>] <тип> <имя формального параметра>

Параметры методов

- Существуют четыре вида параметров, которые специфицируются модификатором:
- *Параметры-значения* – объявляются без модификатора;
- *Параметры, передаваемые по ссылке* – используют модификатор **ref**;
- *Выходные параметры* – объявляются с модификатором **out**; (должно быть присваивание)
- *Параметры-списки* – применяются модификатор **params** (только 1 в конце списка, как массив, каждый – по значению)

Параметры методов

1. Простейшее объявление метода без параметров:

```
void SayHello() {  
    Console.WriteLine("Hello!");  
}
```

2. Метод без аргументов, возвращающий целое значение:

```
int SayInt() {  
    Console.WriteLine("Hello!");  
    return 5;  
}
```

Параметры методов

3. Метод `Add()` выполняет сложение двух аргументов, передаваемых как параметры-значения:

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

4. Метод `ReturnTwo()` возвращает 10 как результат своей работы, кроме этого значение параметра `a` устанавливается равным 100:

```
int ReturnTwo(out int a) {  
    a = 100;  
    return 10;  
}
```

Параметры методов

5. Метод `PrintList()` использует параметр-список:

```
void PrintList(params int[] List) {  
    foreach(int i in List)  
        Console.WriteLine(i);  
}
```

Метод `PrintList()` можно вызвать несколькими способами. Можно передать методу произвольное количество аргументов целого типа или массив целых значений:

```
PrintList(10,20);  
PrintList(1, 2, 3, 4);  
PrintList(new int[] {10, 20, 30, 40});  
PrintList();
```

Параметры методов

При вызове методов на месте формальных параметров помещаются фактические, совпадающие с формальными по типу или приводимые к этому типу. Если при описании параметра использовались модификаторы `ref` или `out`, то они должны быть указаны и при вызове. Кроме этого, фактические параметры с такими модификаторами должны быть представлены переменными, а не литералами или выражениями.

В C# для обмена предусмотрено четыре типа параметров:

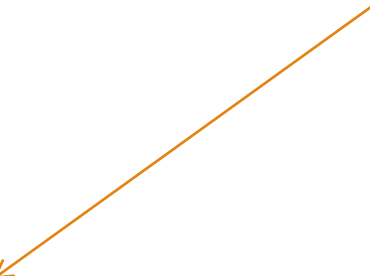
*параметры-значения,
параметры-ссылки,
выходные параметры,
параметры-массивы.*

При передаче параметра по значению метод получает копии параметров, и операторы метода работают с этими копиями

```
class Program
{
    static void Func(int x)
    {
        x += 10;           // изменили значение параметра
        Console.WriteLine("In Func: " + x);
    }

    static void Main()
    {
        int a = 10;
        Console.WriteLine("In Main: " + a);
        Func(a);
        Console.WriteLine("In Main: " + a);
    }
}
```

значение формального параметра x было изменено в методе Func, но эти изменения не отразились на фактическом параметре a метода Main.



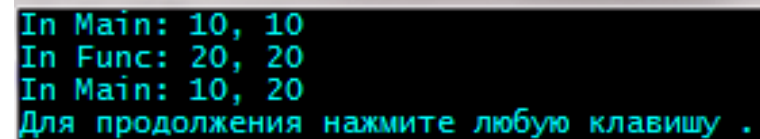
```
In Main: 10
In Func: 20
In Main: 10
Для продолжения нажмите любую клавишу . . . _
```

При передаче параметров по ссылке метод получает копии адресов параметров, что позволяет осуществлять доступ к ячейкам памяти по этим адресам и изменять исходные значения параметров(**ref**)

```
class Program
{
    static void Func(int x, ref int y)
    {
        x += 10; y += 10; //изменение параметров
        Console.WriteLine("In Func: {0}, {1}", x, y);
    }

    static void Main()
    {
        int a=10, b=10; // строка 1
        Console.WriteLine("In Main: {0}, {1}", a, b);
        Func(a, ref b);
        Console.WriteLine("In Main: {0}, {1}", a, b);
    }
}
```

изменения не отразились на фактическом параметре a, т.к. он передавался по значению, но значение b было изменено, т.к. он передавался по ссылке



```
In Main: 10, 10
In Func: 20, 20
In Main: 10, 20
Для продолжения нажмите любую клавишу .
```


Если невозможно инициализировать параметр до вызова метода. Тогда параметр следует передавать как выходной, используя спецификатор **out**

```
class Program
```

```
{  
    static void Func(int x, out int y)  
    {  
        x += 10; y = 10; // определение значения выходного параметра y  
        Console.WriteLine("In Func: {0}, {1}", x, y);  
    }  
}
```

в методе Func формальный параметр y и соответствующий ему фактический параметр b метода Main помечены спецификатором **out**, поэтому значение b до вызова метода Func можно было не определять, но изменение параметра y отразилось на изменении значения параметра b

```
static void Main()  
{  
    int a=10, b;  
    Console.WriteLine("In Main: {0}", a);  
    Func(a, out b);  
    Console.WriteLine("In Main: {0}, {1}", a, b);  
}
```

```
In Main: 10  
In Func: 20, 10  
In Main: 10, 10  
Для продолжения нажмите любую клавишу
```

Классы

СТАТИЧЕСКИЕ КЛАССЫ И СТАТИЧЕСКИЕ ЧЛЕНЫ
КЛАССА

Статические классы

Статический класс в основном такой же, как и нестатический класс, но имеется одно отличие: нельзя создавать экземпляры статического класса. Другими словами, нельзя использовать оператор **new** для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса.

Статические классы

Например, в библиотеке классов .NET статический класс **System.Math** содержит методы, выполняющие математические операции, без требования сохранять или извлекать данные, уникальные для конкретного экземпляра класса **Math**.

Статические классы

Как и в случае с типами всех классов, сведения о типе для статического класса загружаются средой выполнения .NET, когда загружается программа, которая ссылается на класс. Программа не может точно указать, когда загружается класс. Однако гарантируется загрузка этого класса, инициализация его полей и вызов статического конструктора перед первым обращением к классу в программе.

Статические классы

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}
```

<https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>

Статические члены класса

Нестатический класс может содержать статические методы, поля, свойства или события. Статический член вызывается для класса даже в том случае, если не создан экземпляр класса.

Доступ к статическому члену всегда выполняется **по имени** класса, а не экземпляра.

Статические члены класса

Существует **только одна** копия статического члена, независимо от того, сколько создано экземпляров класса.

Статические члены класса

Статические методы и свойства **не могут** обращаться к нестатическим полям и событиям в их содержащем типе, и они не могут обращаться к переменной экземпляра объекта, если он не передается явно в параметре метода.

Классы

СТРУКТУРЫ

Структуры

Тип структуры представляет собой **тип значения**, который может инкапсулировать данные и связанные функции.

Для определения типа структуры используется ключевое слово **struct**

Структуры

Как правило, экземпляр типа структуры создается посредством оператора **new**.

Если все поля экземпляров типа структуры доступны, можно также создать его экземпляр без оператора **new**. В этом случае необходимо инициализировать все поля экземпляров перед первым использованием экземпляра.

Структуры

```
public struct Coords
{
    public double x;
    public double y;
}

public static void Main()
{
    Coords p;
    p.x = 3;
    p.y = 4;
    Console.WriteLine($"{p.x}, {p.y}"); // output: (3, 4)
}
```

<https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct#instantiation-of-a-structure-type>

Структуры

Первое различие между ссылочными типами и типами-значениями, которое мы рассмотрим, заключается в том, что ссылочные типы размещаются в куче и удаляются сборщиком мусора, тогда как типы-значения размещаются либо в стеке, либо встроены в содержащие типы и освобождаются, когда стек раскручивается или когда тип освобождается.

Следовательно, выделение и освобождение типов значений обычно дешевле, чем выделение и освобождение ссылочных типов.

Структуры

Следующее отличие связано с использованием памяти.

Типы значений упаковываются при приведении к ссылочному типу или одному из интерфейсов, которые они реализуют. Они распаковываются, когда возвращаются к типу значения. Поскольку блоки — это объекты, которые размещаются в куче и удаляются сборщиком мусора, слишком частое упаковывание и распаковка может негативно сказаться на куче, сборщике мусора и, в конечном счете, на производительности приложения.

Напротив, такой упаковки не происходит при приведении ссылочных типов.

Структуры

Присваивание ссылочного типа копирует ссылку, тогда как присваивание типа значения копирует значение целиком. Следовательно, присваивания больших ссылочных типов дешевле, чем присваивания больших типов значений.

Структуры

Наконец, ссылочные типы передаются по ссылке, тогда как значимые типы передаются по значению. Изменения экземпляра ссылочного типа влияют на все ссылки, указывающие на экземпляр. Экземпляры типа значения копируются, когда они передаются по значению. Когда экземпляр типа значения изменяется, это, конечно, не влияет ни на одну из его копий. Поскольку копии не создаются пользователем явно, а создаются неявно, когда передаются аргументы или возвращаются возвращаемые значения, типы значений, которые можно изменить, могут сбивать с толку многих пользователей. Следовательно, типы значений должны быть неизменяемыми.