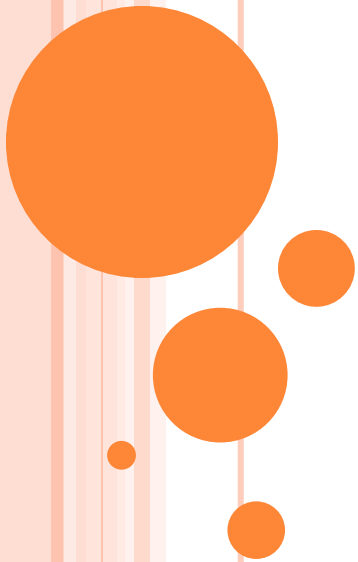


# РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ .NET

Лекция 2

Объектно-ориентированное  
программирование




# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- Наследование в C#
  - Особенности ООП в C#
  - Тип object
- Полиморфизм в C#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в C#
- Дополнительный сведения о структурах и классах
- Преобразование типов





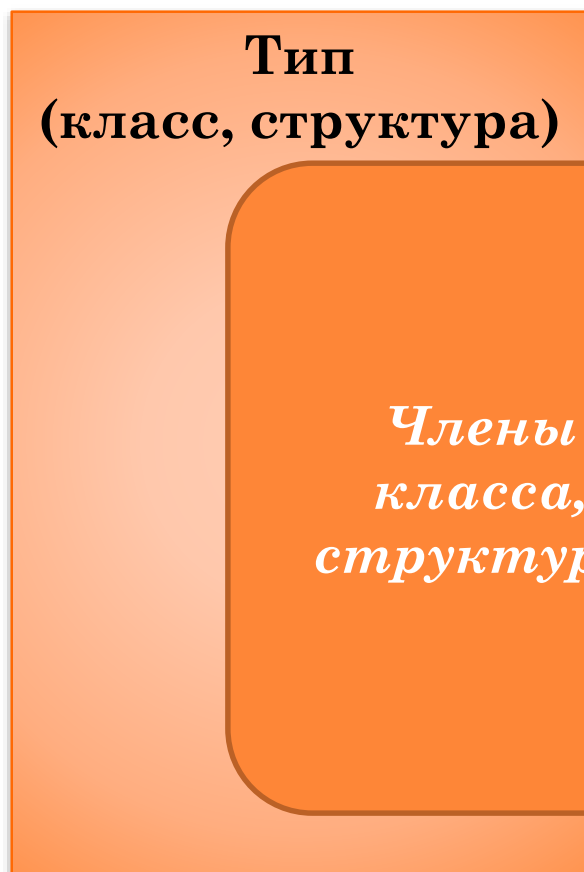
# ОСНОВЫ ОБЪЕКТНО- ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

# ЭВОЛЮЦИЯ

- Линейный код.
- Макросы – повторное использование кода (копируются в указанное место)
- Подпрограммы – повторное использование кода, абстрагирование от основной программы
- Модули (в понимании С).
- Объекты.
  - Совокупность данных, способов преобразования данных, операции с данными



# КЛАСС И ЭКЗЕМПЛЯР КЛАССА



## *Члены класса*

- Поле – содержит данные
- Константы
- Метод – выполняют действия (над данными)
- Свойство (C#) – виртуальное поле (совокупность методов get и set)
- Конструктор – метод, автоматически вызываемый при создании объекта
- Деструктор (финализатор) – метод, автоматически вызываемый при удалении объекта

Объект – это экземпляр типа (класса, структуры)



# КЛАСС. ОБЪЯВЛЕНИЕ В C#

```
[attributes] [modifiers]
  class class_name
  [: [base_class] [,interfaces] ]
{
  [class_member1]
  ...
  [class_memberN]
}
```

Все члены класса описываются и реализуются внутри {}. За пределами **class** ... {...} описывать и реализовывать члены класса нельзя.



# СТРУКТУРА. ОБЪЯВЛЕНИЕ В C#

```
[attributes] [modifiers]
  struct class_name
  [: [interfaces] ]
{
  [class_member1]
  ...
  [class_memberN]
}
```

Все члены структуры описываются и реализуются внутри {}. За пределами **struct** ... {...} описывать и реализовывать члены структуры нельзя.



# ПРИМЕР КЛАССА

```
public class Пиво : Выпивка
{
    public Пиво (string сорт, float градус)
    {
        this.сорт = сорт; this.градус = градус;
    }

    public void Выпить ()
    {
        Console.WriteLine("Ух ты! " + сорт +
            " - жжотъ!!!");
    }

    public string Сорт { get {return сорт;} }
    public float Градус {get {return градус;} }

    protected readonly string сорт;
    protected float градус;
}
```





# СОЗДАНИЕ ОБЪЕКТА

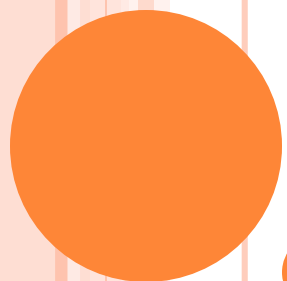
```
class Program
{
    static void Main(string[] args)
    {
        Пиво пивоБалтика = new Пиво("Балтика", 0.05f);

        Console.WriteLine(пивоБалтика.Сорт);
        Console.WriteLine(пивоБалтика.Градус);
        пивоБалтика.Выпить();
    }
}
```

Оператор **new**

- выделяет память под объект
- инициализирует выделенную память 0
- вызывает конструктор (сначала конструкторы базовых типов)
- возвращает ссылку на объект (class) или сам объект (struct).





# ЧЛЕНЫ КЛАССОВ И СТРУКТУР

# ЧЛЕНЫ КЛАССА

- Поле – содержит данные класса
- Константы – определяют “магические” величины
- Метод – выполняют действия (над данными)
- Свойство (C#) – виртуальное поле (совокупность методов get и set)
- Конструктор – метод, автоматически вызываемый при создании объекта
- *Деструктор (финализатор) – метод, автоматически вызываемы при удалении объекта*
- Индексаторы – позволяют работать с объектом как с массивом
- Методы переопределения операций
- События
- Вложенные типы



# ПОЛЕ

- Содержит данные
- Синтаксис

```
[attributes] [modifiers]  
field_type field_name [=initial_value] [,  
    field_nameN [=initial_valueN] ];
```

- Примеры

```
private DateTime dateOfBirth;  
protected object obj = new object();  
private int i=5, j, k=18;  
public readonly double x, y, z;  
private static int nObjects;
```



# ДОСТУП К ПОЛЮ

## ○ Доступ к полям

```
class Vector
{
    public double x;           // Никогда так не делайте.
    public double y;           // Поля практически всегда делаются не public
    public void SetYAsX()
    {
        y = x;                 // Обращение внутри класса как с обычной переменной
    }
}

class Program
{
    static void Main(string[] args)
    {
        Vector v = new Vector();
        v.x = 10;               // Обращение снаружи класса как имя_объекта.имя_поля
        double d = v.x * (v.y + 22); // сможет присутствовать и в левой и правой части выражения
        Console.WriteLine(v.x);    // Может передаваться в методы
    }
}
```



# МОДИФИКАТОР ПОЛЯ: READONLY

- задает поле, доступное только для чтения.
- Поле **readonly** можно задать только непосредственно при объявлении или в конструкторе.
- Изменение такого поля вне конструктора запрещено.
- Пример:

```
class A
{
    private readonly int j = 8;
    private readonly int i;
    public A(int k) { i = k; }
    public TryChange(int k) { i=k; } // Ошибка
}
```

- Использование
  - Как обычную переменную, но только в правой части выражения



# КОНСТАНТЫ

## ○ Синтаксис

- `[modifiers] const type_name const_name = const_expr;`

- Константы могут быть только простых типов и строковые
- Поэтому их применение ограничено – используются для обозначения «магических чисел»
- Члены-константы всегда статические
- Использование – как обычные (статические) поля (только чтение)



# ОБЪЯВЛЕНИЕ МЕТОДА

## ○ Синтаксис:

*[attributes] [modifiers]*

*return\_type method\_name (param\_list) {method\_body}*

## ○ Примеры:

- **public int** ExecuteNonQuery() { ... }

- **public int** Add(**object** obj) { ... }

- **private void** Init(**int** x, **int** y) { ... }

- **public static void** WriteLine(string, **params** **object**[] p) { ... }

## ○ **void** – ключевое слово, для обозначения отсутствия возвращаемого параметра метода

- **private void** Init() { ... }





# ПРИМЕРЫ МЕТОДОВ

```
class Calculator
```

```
{  
    public int Div(int x, int y)  
    {  
        if (y == 0) return 0;  
        return x / y;  
    }  
}
```

```
class Program
```

```
{  
    static void Main(string[] args)  
    {  
        if (args == null || args.Length == 0) return;  
        for (int i = 0; i < args.Length; i++)  
            Console.WriteLine(args[i]);  
        // return не указан  
    }  
}
```

Ключевое слово **return** прекращает выполнение метода и возвращает значение.

Если метод с возвращаемым значением:

Тип возвращаемого значения должен соответствовать типу, описанному в заголовке метода

Все ветки кода должны возвращать значение

Если метод ничего не возвращает (в заголовке метода указано `void`)

Указывается просто `return` без параметров;

`return` можно не указывать



# ВЫЗОВ МЕТОДА

- Синтаксис:
  - `expression.Metod_name(actual_params)`
  - Вызов статического метода
    - `Type_name.Method_name(actual_params)`
- Примеры:
  - `double d = Math.Log(x);`
  - `int j = rnd.Next(0, 10);` // rnd - переменная
  - `Console.WriteLine("Hello, World!");`
- В экземплярный метод неявно передается ссылка на сам объект - `this`, в статический метод – ссылка не передается



# МОДИФИКАТОР STATIC

- **static** – член уровня класса, а не экземпляра класса. Единый для всех экземпляров данного класса
- К статическому члену можно обращаться, не создавая ни одного экземпляра данного класса
- Обращение к статическому члену
- Имя\_типа.Член\_Типа
  - Пример: Console.WriteLine(...)
- При реализации статический член может использовать только статические члены.


```
class Count
{
    public static int number = 0;
    // Внутри типа можно указывать кратко
    // Член_типа, а не Имя_типа.Член_типа
    public void Inc() { ++number; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Count c = new Count();
        c.Inc();
        Console.WriteLine(Count.number);
        Count c2 = new Count();
        c2.Inc();
        Console.WriteLine(Count.number);
    }
}
```

**Результат**

1

2



# ОБЪЯВЛЕНИЕ КОНСТРУКТОРА

- Специальный метод. Вызывается при создании объекта

- Синтаксис:

```
[attributes] [modifiers]
```

```
class name (param_list) [:ctor_call] { ctor_body }
```

```
ctor_call ::= base (actual_params) | this (actual_params)
```

- Примеры:

```
class Vector3d{
```

```
    public Vector3d(double x, double y, double z) {X = x; Y = y; Z = z;}
```

```
    public Vector3d(double x) : this (x, x, x) {}
```

```
}
```

```
class Square : Figure {
```

```
    public Square(int side) : base (side, side) {}
```

```
}
```

- При отсутствии конструкторов у не абстрактного класса, Visual Studio добавляет публичный конструктор без параметров
- Структура всегда имеет конструктор без параметров и его нельзя переопределить
- Любой конструктор у структуры обязан инициализировать все поля структуры
- При генерации исключения объект все равно будет создан
- При создании объекта будут по цепочки вызваны конструкторы базовых типов



# КЛЮЧЕВОЕ СЛОВО THIS

- Обозначает ссылку на текущий **объект**
- Используется:
  - В конструкторах (для вызова другого конструктора этого же типа)

```
class Vector3d{  
    public Vector3d(double x) : this (x, x, x) {}  
    public Vector3d(double x, double y, double z) {}...}
```

- При передачи текущего объекта как параметр  
Print(this);
  - При обращении к членам текущего объекта (при сокрытии)
  - class Customer{  
 string name;  
 void Customer (string name) {this.name = name} ...}
- **this** не может использоваться в статических членах
  - Доступен как в классах так и в структурах



# КЛЮЧЕВОЕ СЛОВО BASE

- Обозначает ссылку на текущий родительский класс
- Используется

- В конструкторах

```
class Vector3d : Vector {
```

```
    public Vector3d(double x) : base(x, x, x) {}...}
```

- При вызове функционала базового класса

```
public override void Print (string text)
```

```
{
```

```
    ...
```

```
    base.Print(text + “ еще и мой текст”);
```

```
}
```



# СТАТИЧЕСКИЙ КОНСТРУКТОР

- Назначение – выполнение действий по инициализации **типа**
- Выполняется до обращения к любому статическому члену и до создания первого объекта
- Статические конструкторы различных типов выполняются в произвольном порядке

- Синтаксис:

```
static class_name() {  
    ctor_body  
}
```

- Не может иметь модификатор доступа



# СВОЙСТВО

- Свойство – виртуальное поле
  - Обращение – как к полю:
    - `int len = s.Length;`
    - `page.Title = "Hello, World!";`
  - Реально при обращении вызывается метод – **аксессор**
- Поле может иметь аксессор для
  - чтения (**get**)
  - записи (**set**)





# ОБЪЯВЛЕНИЕ СВОЙСТВА

- Синтаксис:

```
[attributes] [modifiers]
  prop_type prop_name {
    [[access_modifier] get accessor_body]
    [[access_modifier] set accessor_body]
  }
```

- Пример:

```
public int[] Numbers {
  get
  {
    if(ns == null) ns = {1, 2, 3, 4, 5};
    return ns;
  }
  private set { ns = value; }
}
```

- **get** вызывается при получении значения свойства  
`int[] variable = meObject.Numbers;`
- **set** вызывается при установке значения свойства  
`meObject.Numbers = variable;`
- **value** – ключевое слово в блоке `set`, обозначающее полученное значение
- Блок `set` – может отсутствовать – получится свойство только для чтения
- Блок `get` – может отсутствовать – получится свойство только для установки значения (такой вариант практически не используется)



# ИСПОЛЬЗОВАНИЕ СВОЙСТВ

- Для доступа к полям используют свойства
  - Практически всегда Поля – `private/protected`. Для соблюдения инкапсуляции.
  - Даже если сейчас нет никакой логики в `get`, `set`, она может появиться завтра
- Ограничение доступа к полям
  - Поля только для чтения
  - Проверка допустимости нового значения
- Вычисляемые «поля»
- Ленивая инициализация
- Представление одного поля в нескольких форматах



# АВТОМАТИЧЕСКИЕ СВОЙСТВА

- Синтаксис:

```
[attributes] [modifiers]
  prop_type prop_name {
    [access_modifier] get;
    [access_modifier] set;
  }
```

- Пример:

```
public int[] Numbers {get; set;}
public DateTime birthday {get; private set;}
```

За кулисами будет создано `private` поле. `get` – будет возвращать значение этого поля, а `set` – устанавливать его

Тело и `get`, и `set` должно отсутствовать при описании автоматического свойства



# АВТОМАТИЧЕСКИЕ СВОЙСТВА С ПЕРВОНАЧАЛЬНОЙ ИНИЦИАЛИЗАЦИЕЙ

- Свойство с первоначальной инициализацией :
  - `public string User { get; set; } = WindowsIdentity.GetCurrent().Name;`
- Свойства только для чтения с read-only-defined backing field. У автоматического свойства не задается set
  - `public DateTime TimeStamp { get; } = DateTime.UtcNow;`
  - `public string User { get; } = WindowsIdentity.GetCurrent().Name;`
  - `public string ProcessName { get; } = Process.GetCurrentProcess().ProcessName;`
- Автоматическое readonly свойство можно задать в конструкторе

```
class Temperature {  
    public int TempInK { get; }  
    public Temperature() { TempInK = 273; }  
}
```



# ИНДЕКСАТОР

- Индексатор – свойство, используемое для перегрузки операции [ ]
- В основном используется с различными коллекциями
- Индексатор не может быть статическим
- Синтаксис

```
[attributes] [modifiers]  
ind_type this [param_list] {  
  [[access_modifier] get accessor_body]  
  [[access_modifier] set accessor_body]  
}
```

- Пример:

```
public float this[string koord]  
{  
    get { if(koord ==“X”) return fx; else return fy; }  
    set { if(koord ==“X”) fx = value; else fy = value;}  
}
```

- Обращение

```
float real = vector[“X”];  
vector[“Y”] = 10;
```



# ПЕРЕГРУЗКА ФУНКЦИЙ (OVERLOADING)

- **Перегрузка функций** – объявление нескольких функций с одинаковым именем и разной сигнатурой
  - **Сигнатура функции** включает имя функции, а также список формальных параметров
  - Сигнатура **не включает** возвращаемого значения!
- **Примеры:**
  - `int Inc(int i) {...}`
  - `double Inc (double d) {...}`
  - `int Inc(int i, long j ) {...}`
  - `double Inc (int d) {...} // Ошибка`



# ПЕРЕГРУЗКА ОПЕРАЦИЙ

- В отличие от некоторых языков, C# позволяет переопределять операции
- Переопределять можно
  - Арифметические операции
    - Унарные
    - Бинарные
  - Операции приведения типов
- Методы, перегружающие операцию должны быть статическими



# ПЕРЕГРУЗКА АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

- Унарные операции

```
public static return_type operator operation(source_type param_name)
```

- Бинарные операции

```
public static return_type operator operation(source_type param_name, source_type2 param_name2)
```

- Хотя бы один из двух параметров должен совпадать с типом, содержащим оператор
- Методы обязательно статические
- Пример

```
public static Vector operator +(Vector a, Vector b)
{
    return new Vector(a.fx + b.fx, a.fy + b.fy);
}
```





# ОПРЕДЕЛЕНИЕ ОПЕРАТОРОВ ПРИВЕДЕНИЯ ТИПА

- Явное приведение

```
public static explicit operator  
    result_type(source_type param_name)
```

- Неявное приведение

```
public static implicit operator  
    result_type(source_type param_name)
```

- Один из двух типов должен совпадать с типом, содержащим оператор

- Примеры:

```
public static implicit operator Vector3D(int x) {  
    return new Vector3D(x, 0, 0); }  
public static explicit operator float(Vector3D v) {  
    return v.fx; }
```



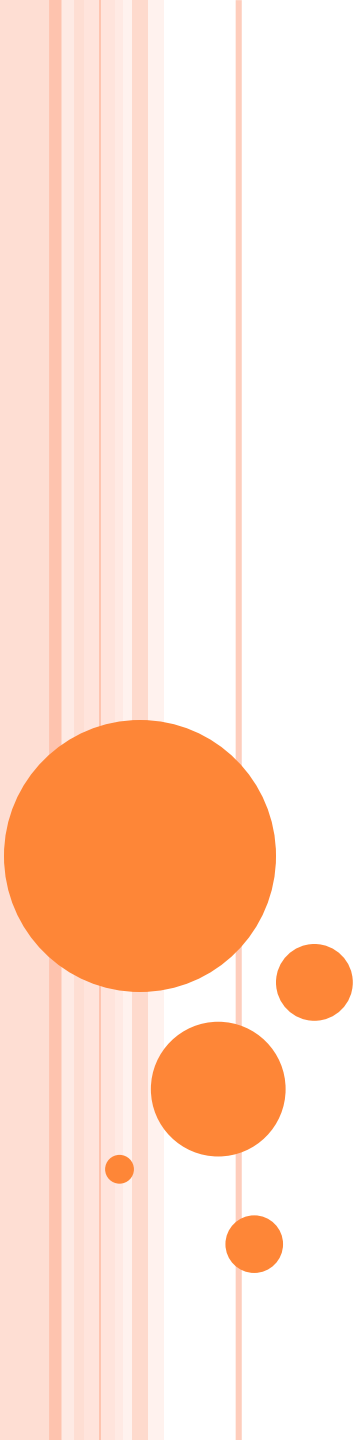
# ПЕРЕГРУЖАЕМЫЕ ОПЕРАТОРЫ

Операторы	Возможность перегрузки
+, -, !, ~, ++, --, true, false	Унарные операторы могут быть перегружены.
+, -, *, /, %, &,  , ^, <<, >>	Бинарные операторы могут быть перегружены.
==, !=, <, >, <=, >=	Операторы сравнения могут быть перегружены, но парами
&&,	Условные логические операторы не могут быть перегружены, но они оцениваются с помощью & и  , которые могут быть перегружены.
[]	Оператор индексирования массива не может быть перегружен, но можно определить свои индексаторы.
(T)x	Оператор приведения типов не может быть перегружен, но можно определить новые операторы преобразования (explicit и implicit).
+=, -=, *=, /=, %= &=,  =, ^=, <<=, >>=	Операторы присваивания не могут быть перегружены явным образом. Однако при перегрузке бинарного оператора соответствующий оператор присваивания (если таковой имеется) также неявно перегружается. Например, += вычисляется с помощью +, который может быть перегружен.
=, .., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	Эти операторы не могут быть перегружены.

# ЧЛЕНЫ КЛАССА

- Поле – содержит данные класса
- Константы – определяют “магические” величины
- Метод – выполняют действия (над данными)
- Свойство (C#) – виртуальное поле (совокупность методов get и set)
- Конструктор – метод, автоматически вызываемый при создании объекта
- *Деструктор (финализатор) – метод, автоматически вызываемы при удалении объекта*
- Индексаторы – позволяют работать с объектом как с массивом
- Методы переопределения операций
- События
- Вложенные типы





# **ОСНОВНЫЕ ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ (ООП)**

# ОСНОВНЫЕ ПРИНЦИПЫ ООП

## ○ Инкапсуляция

- Скрытие реализации.
  - Отделение интерфейса от реализации.
  - Работа с данными только через методы
- Например:

```
string s = Console.ReadLine()
```

## ○ Наследование

- Расширение функциональности базового класса в производном (дочернем) классе
- Многократное использование кода

## ○ Полиморфизм

- Сопоставление одному имени нескольких сущностей
- Одно имя – несколько реализаций



# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- **Наследование в C#**
  - Особенности ООП в C#
  - Тип object
- Полиморфизм в C#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в C#
- Дополнительный сведения о структурах и классах
- Преобразование типов



# НАСЛЕДОВАНИЕ

- Расширение функциональности базового класса
- Повторное использование кода

```
public class Выпивка
{
    public void УгоститьДруга(string имяДруга) { Console.WriteLine("Спасибо. " + имяДруга + " порадовался"); }
}
```

```
public class Пиво : Выпивка
{
    public Пиво(string сорт, float градус) { Сорт = сорт; }
    public void Выпить() { Console.WriteLine("Ух ты! " + Сорт + " – жжоть!!!"); }
    public string Сорт { get; private set; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Пиво пивоБалтика = new Пиво("Балтика", 5);
        пивоБалтика.УгоститьДруга("Петя");
        пивоБалтика.Выпить();
    }
}
```



# СИНТАКСИС НАСЛЕДОВАНИЯ

[Атрибуты] [Модификаторы]

**class** *ИмяПроизводногоКласса* : *ИмяБазовогоКласса*

{

*Тело класса*

}

- **Базовый (родительский) класс** – тот класс, от которого наследуют
- **Производный класс** – тот, который наследует
- Базовый и производный класс могут быть написаны на разных языках

```
class Matrix { public int Rang() { ... }; }  
class DiagMatrix : Matrix { public bool isE()  
    { ... }; }  
DiagMatrix dm = new DiagMatrix();  
int i = dm.Rang();
```





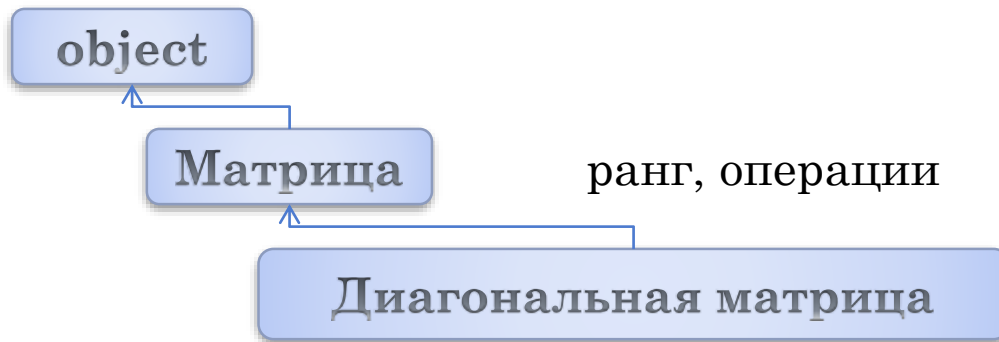
# ОСОБЕННОСТИ НАСЛЕДОВАНИЯ В C#

- Запрещено множественное наследование.
  - Если необходимо унаследовать от множества типов, можно использовать принцип “has a” и делегировать необходимые методы вложенного типа. Делегирование.
- Нет модификаторов наследования.
  - Наследование всегда public.
- Базовый и производный класс могут быть написаны на разных языках
- Наследование структур запрещено.



# НАСЛЕДОВАНИЕ

- Отношение “is a” – является.



Традиционное наследование

- Отношение “has a” – содержит.



Делегирование

Workaround при необходимости множественного наследования



# РЕАЛИЗАЦИЯ ПРИНЦИПА “HAS A”

```
public class Audio
{
    public void Play(Song song)
    {
        Console.WriteLine("Исполняется {0}", song);
    }
}
```

```
public class Car
{
    private Audio coolAudio = new Audio();

    public void Play(Song song)
    {
        coolAudio.Play(song);
    }
}
```

Делегирование



# ИЕРАРХИЯ КЛАССОВ

- C++ - ациклический ориентированный граф
- C# - дерево
- Корнем дерева в C# служит тип `System.Object`
  - или сокращенно просто **object**

При описании типа

```
class Car // неявное наследование от класса object
{
}
```



# МЕТОДЫ SYSTEM.OBJECT

- string **ToString()**
  - Позволяет получить строковое представление объекта
  - По умолчанию возвращает квалифицированное имя типа
- int **GetHashCode()**
  - Позволяет получать хэш-код объекта
- bool **Equals(object o)**
  - Позволяет сравнивать любые объекты
  - Для ссылочных типов по умолчанию сравнение на равенство ссылок
  - Для типов значения по умолчанию сравнение значений
- Type **GetType()**
- Переопределяйте, если семантика по умолчанию неприемлема (кроме GetType())



# СТАТИЧЕСКИЙ И ДИНАМИЧЕСКИЙ ТИП

- Переменная базового класса может ссылаться на объект производного класса
- **Статический тип** – тип переменной.
- **Динамический тип** – реальный тип объекта, на который переменная указывает.
- **Пример:**
  - `Control ctrl = new Button("OK");`
  - Статический тип – `Control`
  - Динамический тип - `Button`



## МЕТОД ОБЪЕКТ . GETTYPE ( )

- Метод `GetType ( )` возвращает объект типа `System.Type`
- Возвращаемый объект соответствует динамическому типу объекта
- Метод `GetType ( )` нельзя переопределить
- На этом держится вся типобезопасность в .NET



# НЕКОТОРЫЕ СПЕЦИАЛЬНЫЕ БАЗОВЫЕ ТИПЫ

- System.**Array** : object
  - базовый класс для всех массивов
- System.**ValueType** : object
  - базовый класс для всех типов-значений
- System.**Enum** : System.ValueType
  - базовый класс для всех перечислений
- System.**Exception** : object
  - базовый класс для всех исключений
- System.**Delegate** : object
  - базовый класс для всех делегатов





# ПРЕОБРАЗОВАНИЯ ТИПОВ

- Ссылка на производный класс **неявно** приводится к ссылке на базовый класс

```
Control ctrl = new Button("OK");
```

- Ссылка на базовый класс может быть **явно** приведена к ссылке на производный класс

```
Button button = (Button)ctrl;
```

- Если подобное преобразование недействительно, то выбрасывается исключение

- Функция, принимающая базовый класс в качестве параметра, может принять и его производный класс

```
void Show(Control ctrl){...}
```

```
Show(button);
```



# НАСЛЕДОВАНИЕ

## ○ Добавление новой функциональности

- Объявление новых членов

```
class Matrix
```

```
    { public int Rang() { ... }; }
```

```
class DiagMatrix : Matrix
```

```
    { public bool isE() { ... };
```

```
...
```

```
DiagMatrix dm = new DiagMatrix();
```

```
int i = dm.Rang();
```

```
bool b = dm.isE();
```

## ○ Изменение базовой функциональности

- Перекрытие базовых членов
- Переопределение (overriding) базовых методов



# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- Наследование в C#
  - Особенности ООП в C#
  - Тип object
- Полиморфизм в C#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в C#
- Дополнительный сведения о структурах и классах
- Преобразование типов



# ПОЛИМОРФИЗМ

## Использование связанных объектов одинаковым образом

```
class Shape
{
    // Рисует фигуру
    public virtual void Draw() { ... }
}

class Circle : Shape
{
    // Рисует окружность
    public override void Draw() { ... }
}

class Hexagon : Shape
{
    // Рисует многоугольник
    public override void Draw() { ... }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Shape shape = new Shape();
        shape.Draw();
        shape = new Circle();
        shape.Draw();
        shape = new Hexagon();
        shape.Draw();
    }
}
```



# ПОЛИМОРФИЗМ

- Динамический полиморфизм
  - Виртуальные члены
  - Абстрактные члены
- Статический полиморфизм
  - Переопределение функций (overloading)
  - Переопределение операций
  - Соккрытие (hiding) членов



# ВИРТУАЛЬНЫЕ ЧЛЕНЫ

- Виртуальные члены позволяют обеспечивать различные реализации

```
class Shape {public virtual void Draw() { }}
```

```
class Circle : Shape {public override void Draw() { }}
```

```
class Hexagon : Shape{public override void Draw() { }}
```

- Виртуальные члены обеспечивают связывание на основе динамического типа

```
Shape sh1 = new Circle();      sh1.Draw()
```

```
Shape sh2 = new Shape();      sh2.Draw()
```

```
Shape sh3 = new Hexagon();    sh3.Draw()
```

- Виртуальные члены позволяют изменять функциональность базового класса

- Виртуальные методы должны иметь одну и ту же сигнатуру (т.е. **имя** метода и список **входных** и **выходных** параметров)



# ОБЪЯВЛЕНИЕ ВИРТУАЛЬНЫХ ЧЛЕНОВ

- При объявлении используется модификатор **virtual**
- Виртуальные члены не могут быть статическими или закрытыми
- Примеры виртуальных членов
  - `public virtual int Method();`
  - `public virtual double Property {...}`
  - `protected virtual void OnPaint();`



# ПЕРЕКРЫТИЕ ВИРТУАЛЬНЫХ ЧЛЕНОВ

- Для перекрытия виртуальных членов используется модификатор **override**
- Специальный модификатор необходим для лучшей поддержки IDE
- Примеры перекрытия:
  - **public override string ToString() {}**
  - **protected override void Draw() {}**





# СОКРЫТИЕ ЧЛЕНОВ

- Член (не метод и не индексатор) скрывает все члены с тем же именем в базовом классе

```
class Shape { int Square;}
```

```
class Circle : Shape { double Square; }
```

- Метод скрывает все члены с тем же именем и методы с той же сигнатурой в базовом классе

Необходимо указывать `new` или `override`. По умолчанию `new`.

- Индексатор скрывает индексатор с той же сигнатурой в базовом классе

- Сокрытие **распространяется** вниз по иерархии



# МОДИФИКАТОР NEW

- Если случается сокрытие членов, компилятор выдает предупреждение
- Чтобы избежать этого, используйте модификатор **new**
- **new** помогает разрешить проблемы с версиями
- Сокрытие с **new** распространяется вниз по иерархии
  - `protected override void Draw() {};`
  - `protected new void Draw() {};`



# ПРИМЕР

```
class Program
{
    static void Main(string[] args)
    {
        Shape sh = new Shape(); Console.WriteLine(sh.ShapeName());
        sh = new Hexagon(); Console.WriteLine(sh.ShapeName());
        sh = new Square(); Console.WriteLine(sh.ShapeName());
        sh = new Triangle(); Console.WriteLine(sh.ShapeName());
    }
}
class Shape
{
    public virtual string ShapeName() { return "Фигура"; }
}
class Hexagon : Shape
{
    public override string ShapeName() { return "Многоугольник"; }
}
class Square : Hexagon
{
    public override string ShapeName() { return "Квадрат"; }
}
class Triangle : Hexagon
{
    public new string ShapeName() { return "Треугольник"; }
}
}
```

```
Фигура
Многоугольник
Квадрат
Многоугольник
_
```



# OVERLOADING

- Методы в классе могут иметь одинаковое имя.
- Методы с одинаковым именем обязаны отличаться входными параметрами
- Сигнатура метода – имя метода + список входных параметров. Методы внутри типа должны иметь уникальную сигнатуру.
- Методы отличающиеся только выходным параметром не могут существовать внутри одного типа\*

```
class Calculator
{
    public int Div ( int x,    int y ) { return x / y; }
    public double Div (double x, double y) { return x / y; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        int intResult = calc.Div(11, 5);
        double doubleResult = calc.Div(11.0, 5.0);
    }
}
```

\* В C# не могут, в .NET могут (см. Рихтер)



# АБСТРАКТНЫЙ МЕТОД И АБСТРАКТНЫЙ КЛАСС

- `abstract class A {  
 public abstract void GetName(); }`

- Абстрактный метод не содержит реализации
- Абстрактный метод задают функциональность, которая должна быть реализована в классах потомках.
- Класс имеющих хоть 1 абстрактный метод должен быть объявлен как абстрактный
- Нельзя создать экземпляр абстрактного класса
- Потомки обязаны реализовать абстрактные члены или должны быть сами объявлены как абстрактные.
- Виртуальные методы МОГУТ быть переопределены в потомках, абстрактные – ДОЛЖНЫ быть переопределены в потомках
- Пример:

```
abstract class A {public abstract void GetName(); }  
class B {public override void GetName() { реализация } }
```



# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- Наследование в С#
  - Особенности ООП в С#
  - Тип object
- Полиморфизм в С#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в С#
- Дополнительный сведения о структурах и классах
- Преобразование типов



# ИНКАПСУЛЯЦИЯ

- Соккрытие реализации, деталей
- Достигается с помощью модификаторов доступа
  - **public** – член класса виден всем
  - **protected** – член класса виден только внутри самого класса и внутри потомков
  - **private** – виден только внутри объявившего его класса (*по умолчанию*)
  - **internal** – виден только внутри сборки
  - **protected internal** – виден только внутри сборки и для потомков в любой сборке
  - **private protected** – виден только внутри потомков текущей сборки. *Доступно с версии C# 7.2.*
- Модификаторов доступа применяются также и к типам
  - **public** – класс (тип) виден всем
  - **internal** – класс(тип) виден только внутри сборки (*по умолчанию*)
- В качестве члена класса может выступать и другой тип. Тогда к нему применимы модификаторы доступа к члену класса



# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- Наследование в С#
  - Особенности ООП в С#
  - Тип object
- Полиморфизм в С#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в С#
- **Дополнительный сведения о структурах и классах**
- Преобразование типов







# ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О СТРУКТУРАХ И КЛАССАХ

# ОТЛИЧИЕ КЛАССА ОТ СТРУКТУРЫ

- Большинство синтаксиса и возможностей одинаковые
- Класс- ссылочный тип, структура – тип-значение
- При приравнении структуры значения всех полей копируется, а при приравнении класса – копируется только ссылка на объект
  - Аккуратно с полями ссылочных типов в структурах. Копирование полей – копирование ссылок на объекты
- Поля в структуре не могут быть проинициализированы при описании (не относится к константам и статическим полям)
- Структура не может переопределять конструктор по умолчанию (без параметров) и деструктор.
- В отличие от класса, структура может создаваться без ключевого слова new. Пример: `int i = 5;`
- Структура не может быть явно унаследована и не может является источником наследования. Все структуры неявно унаследованы от типа `System.ValueType`, который унаследован от `object`.
- Структура также как и класс может реализовывать интерфейсы
- Структура может использоваться как `Nullable<T>` тип, который может принимать значения `null`.



# КОНСТРУКТОР ПО УМОЛЧАНИЮ

## ○ Class

- Если не указано ни одного конструктора Visual Studio создает конструктор по умолчанию
  - `public ИмяТупа( ) { }`
- Если задается любой конструктор, конструктор по умолчанию не добавляется

## ○ Struct

- Всегда имеет неявный `public` конструктор по умолчанию и его нельзя переопределить
- Можно заводить дополнительные конструкторы
  - `ComplexStruct myStruct = new ComplexStruct(5,7);`
- Дополнительные конструкторы должны проинициализировать все поля или вызвать конструктор по умолчанию



# МОДИФИКАТОР SEALED

- Модификатор `sealed` для типа
  - От класса, помеченного как `sealed`, нельзя наследоваться
  - Все структуры неявно `sealed`, нельзя наследоваться
  - Использование
    - В классах отвечающих за безопасность
      - Нельзя в потомках получить доступ к незащищенным инкапсулированным данным
      - Нельзя подменить алгоритмы обеспечения безопасности
    - Для повышения быстродействия
      - Отключается поиск переопределения виртуальных методов

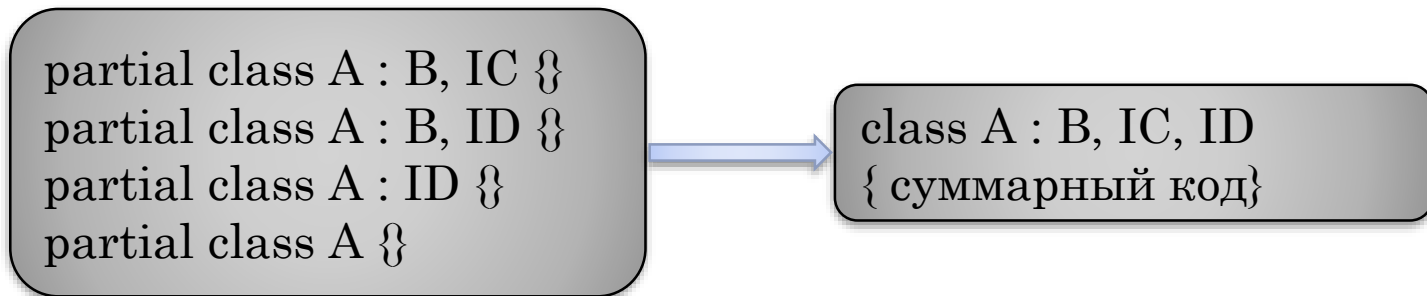
```
public sealed class MySecurity { ... }
```

- Модификатор `sealed` для метода или свойства
  - Если виртуальный метод или свойство помечены как `sealed`, то их нельзя переопределять в наследниках (обрывает цепочку виртуальности)



# МОДИФИКАТОР PARTIAL

- Для класса, структуры и интерфейса
  - partial тип может быть определен в разных файлах
  - Все части partial типа должны быть определены внутри **одного модуля** (сборки)
  - Все partial части должны иметь одинаковое имя
  - Компилятор склеит все partial части типа в один тип при компиляции
  - Строка наследования и реализуемых интерфейсов тоже будет склеена (одинаковые типы удалятся)



Используется для разбиения на части большого сложного типа и реализации разных логических частей в разных файлах

# МОДИФИКАТОР PARTIAL

## ○ Для метода

- Только внутри `partial` класса или структуры
- Метод должен возвращать `void`
- Не допускается использовать модификатор доступа. Он всегда неявно `private`
- Сигнатуры методов должны совпадать
- В одной части должно быть объявление метода

```
partial void OnSomethingHappened(string s);
```

- В другой *может* быть реализация

```
partial void OnSomethingHappened(String s)  
{  
    Console.WriteLine("Something happened: {0}", s);  
}
```

- Если реализации нет, то все вызовы метода удаляются при компиляции



# РАСШИРЕННЫЙ СИНТАКСИС ИНИЦИАЛИЗАЦИИ

После конструктора в { } можно задавать публичные свойства

При этом можно опускать () при вызове конструктора без параметров

```
class Complex
{
    public double Re { get; set; }
    public double Im { get; set; }

    public Complex() {}
    public Complex(double re) { Re = re; }
}
```

...

```
Complex c1 = new Complex() { Re = 5, Im = 7 };
Complex c2 = new Complex() { Im = 7 };
Complex c3 = new Complex { Im = 7 };
Complex c4 = new Complex(5) { Im = 7 };
```



# СЕГОДНЯ

## ○ Основы ООП

- Введение
- Пример класса. Описание, создание экземпляра
- Члены классов и структур
- Основные принципы ООП
- Наследование в С#
  - Особенности ООП в С#
  - Тип object
- Полиморфизм в С#
  - virtual, override, new, abstract class, abstract method
- Инкапсуляция в С#
- Дополнительный сведения о структурах и классах
- Преобразование типов





# ПРЕОБРАЗОВАНИЯ ТИПОВ

- Ссылка на производный класс **неявно** приводится к ссылке на базовый класс

```
Control ctrl = new Button("OK");
```

- Ссылка на базовый класс может быть **явно** приведена к ссылке на производный класс

```
Button button = (Button)ctrl;
```

- Если подобное преобразование недействительно, то выбрасывается исключение

- Функция, принимающая базовый класс в качестве параметра, может принять и его производный класс

```
void Show(Control ctrl){...}
```

```
Show(button);
```



# ПРЕОБРАЗОВАНИЕ ТИПОВ

## ○ Обычный синтаксис:

- *(Имя\_типа) выражение*
- Если преобразование не проходит, то вызывается исключение

```
Button button = (Button)ctrl;
```

## ○ Оператор **as**

- *expression* **as** *Имя\_типа*
- Если преобразование не проходит, то возвращается **null**
- Применимо только к ссылочным типам

```
Button button = ctrl as Button;
```



# ОПЕРАТОР IS

- Оператор `is` проверяет, имеет ли выражение заданный тип

*выражение* **is** *Имя\_типа*

- Оператор возвращает `true`, если
  - Выражение не `null`
  - Преобразование выражения в заданный тип проходит

- Пример

```
if(number is int)
{
    int i = (int)number; ...
}
else if(number is double) {
    double d = (double)number; ...
}
```



# ОПЕРАТОР TYPEOF

- Служит для получения объекта `System.Type` для заданного **типа**
- **typeof** (*Имя\_Типа*)

## Пример

```
if (number.GetType () == typeof (int))  
{ ... }  
else  
    if (number.GetType () == typeof (double))  
    { ... }
```

- Можно использовать для точной проверки динамического типа



# СПЕЦИАЛЬНЫЕ ПРЕОБРАЗОВАНИЯ НЕКОТОРЫХ ТИПОВ

- Преобразование типа string к типам int, long, short, double и т.п.:

*Имя\_типа*.Parse(string)

Например: `int i = int.Parse("5");`

- Безопасное преобразование:

*bool* *Имя\_типа*.TryParse(string, out *Имя\_переменной*)

Например:

```
int i;  
if (int.TryParse("52", out i)) { /*Используем i*/ };
```

В C# 7

```
if (int.TryParse("55", out int j)) { /*Используем j*/ };
```

- Преобразование различных типов значимые типы:

Convert.ToXXX(object)

Например: `int i = Convert.ToInt32('5');`

- Преобразование к строке: метод ToString()

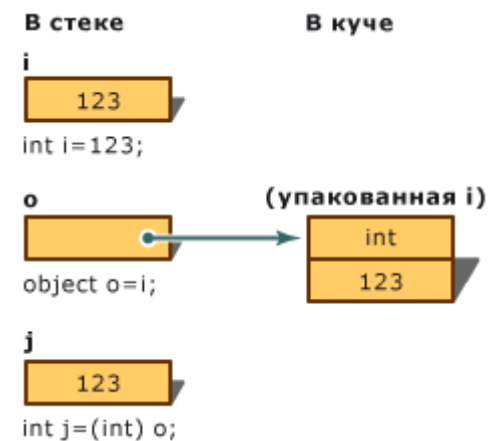
Например: `int i = 5;`

```
string s = i.ToString();
```



# BOXING / UNBOXING

- Происходит при тип приведении типов-значений к `object` или к типу интерфейса
- **boxing**: в куче создается специальный объект, в который упаковывается объект-значение
  - `int i = 123;`
  - `object o = i; // boxing`
- **unboxing**: значение извлекается из этого упакованного объекта
  - `int j = (int)o; // unboxing`
- При упаковке и распаковке создаются **новые объекты**
- Избегайте частых преобразований **boxing / unboxing**.
- Вызывается также при вызове у структуры методов типа `object*`. Например, `ToString()`



\* - если нет переопределения в типе-значения

# ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ В C#7

## СОПОСТАВЛЕНИЕ С ШАБЛОНОМ

### ○ Шаблоны в is

- is null – проверка на null
- is type\_name variable – проверка на тип, приведение типа, присваивание в новую переменную

```
public int Demo(object o)
{
    if (o is null) return 0;    // шаблон "null"
    if (!(o is int i)) return 0; // шаблон типа
    return i * i;
}

public int Demo2(object o)
{
    if (o is int i || (o is string s && int.TryParse(s, out i)))
    {
        return i * i;
    }
    return 0;
}
```



# ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ В C#7

## ШАБЛОНЫ И ВЫРАЖЕНИЯ В SWITCH

- Теперь позволяет использовать любые типы, а не только простые
- Можно использовать шаблоны в выражениях case.
- Ключевое слово *when* - можно добавлять дополнительные условия к выражениям case

```
public void Demo(Shape shape)
{
    switch (shape)
    {
        case Circle c:
            Console.WriteLine($"круг с радиусом {c.Radius}");
            break;
        case Rectangle s when (s.Length == s.Height):
            Console.WriteLine($"{s.Length} x {s.Height} квадрат");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Length} x {r.Height} прямоугольник");
            break;
        default: // всегда вычисляется последним
            Console.WriteLine("неизвестная фигура");
            break;
        case null:
            Console.WriteLine("фигура отсутствует");
            break;
    }
}
```

