Разработка .NET приложений

Лекция 2

Объектно-ориентированное программирование

Сегодня

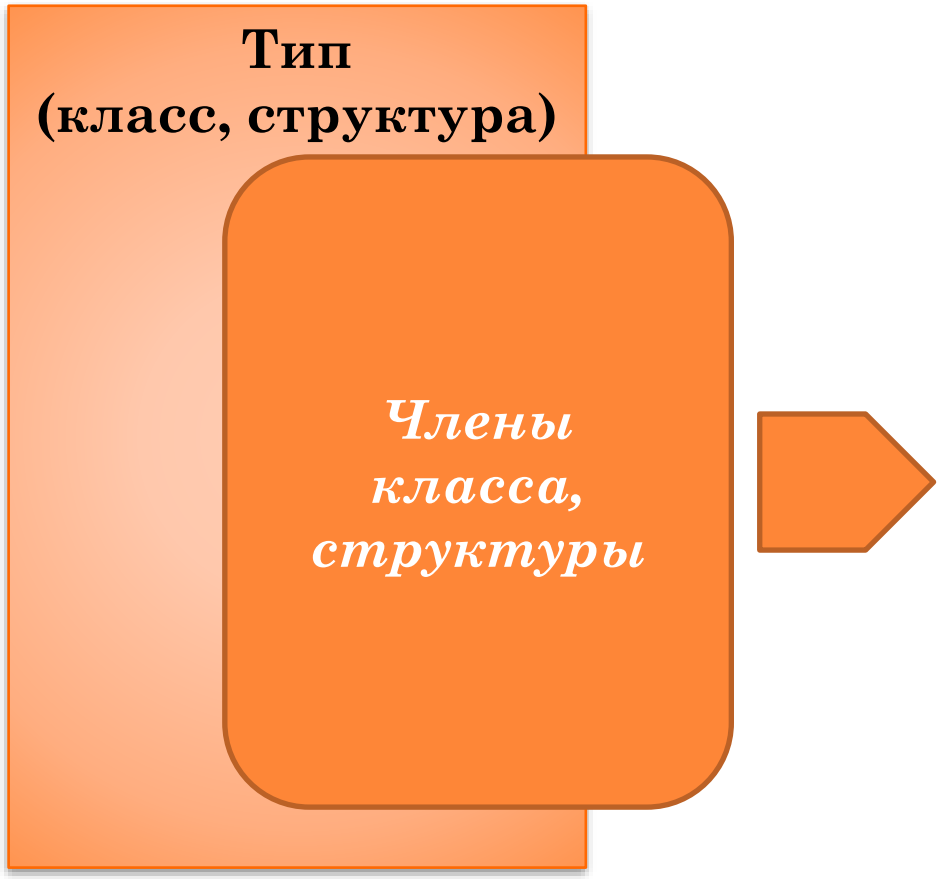
- Основы ООП
  - Введение
  - Пример класса. Описание, создание экземпляра
  - Члены классов и структур
  - Основные принципы ООП
  - Наследование в C#
    - Особенности ООП в C#
    - Тип object
  - Полиморфизм в C#
    - virtual, override, new, abstract class, abstract method
  - Инкапсуляция в C#
  - 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 – выполнят 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 struct_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 Газировка : Напиток
{
  public Газировка (string название, float сахар)
   {
        this.название = название;
        this.сахар = сахар;
   }

  public void Выпить()
   {
       Console.WriteLine("Ух ты! " + название + " – освежает !!!");
   }

  public string Название { get {return название;} }
  public float Сахар {get {return сахар;} }

  protected readonly string название;
  private float сахар;
}
```

Создание объекта

```
class Program
{
   static void Main(string[] args)
   {
      Газировка лимонад = new Газировка("Тархун", 0.05f);

      Console.WriteLine(лимонад.Название);
      Console.WriteLine(лимонад.Сахар);
      лимонад.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 – выполнят 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*
- 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 справой 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 void 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
- Значения должны быть доступны в процессе компиляции
- 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 s, 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 (в C# 12 можно)
- 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 обращение к членам текущего объекта (при сокрытии)
  - class Customer{

```
    string name;
    public 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**) или для инициализации (**init**)

Объявление свойства

- Синтаксис:

```
[attributes] [modifiers]
 prop_type prop_name {
  [[access_modifier] get
  accessor_body]
  [[access_modifier] [set|init]
  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
- Вычислимые «поля»
- Ленивая инициализация
- Представление одного поля в нескольких форматах

Автоматические свойства

- Синтаксис:

```
[attributes] [modifiers]
 prop_type prop_name {
   [access_modifier] get;
   [access_modifier] set;
   [access_modifier] init; // может быть только что-то одно или set, или init
}
```

- Пример:

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

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 и в конструкторе

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

- Автоматическое readonly свойство, которое можно задавать из конструктора и с помощью расширенного синтаксиса инициализации

```
class Complex{
        public double Real { get; init; };
} …

Complex complex = new Complex() { Real = 10.0 };
```

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 методов** – объявление нескольких методов с одинаковым именем и разной сигнатурой
  - **Сигнатура метода** включает в себя имя метода, 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

* C#, но не .NET. .NET допускает создание несколько методов, отличающихся только выходными параметрами

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 – выполнят 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**
  - Сокрытие 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

```
class Program
  {
    static void Main(string[] args)
    {
            Газировка напиток = new Газировка(“Байкал");

            напиток.УгоститьДруга("Петя");

             напиток.Выпить();
    }
  }
```

```
public class Вода
  {
    public void УгоститьДруга(string имяДруга)
    {
        Console.WriteLine("Спасибо. " + имяДруга
                                  + " не дал умереть от жажды");
    }
  }

  public class Газировка : Вода
  {
    public Газировка(string название)
    {
        Название = название;
    }
    public void Выпить()
    {
        Console.WriteLine("Ух ты! " + Название + " – освежает !!!");
    }
    public string Название { get; }
  }
```

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

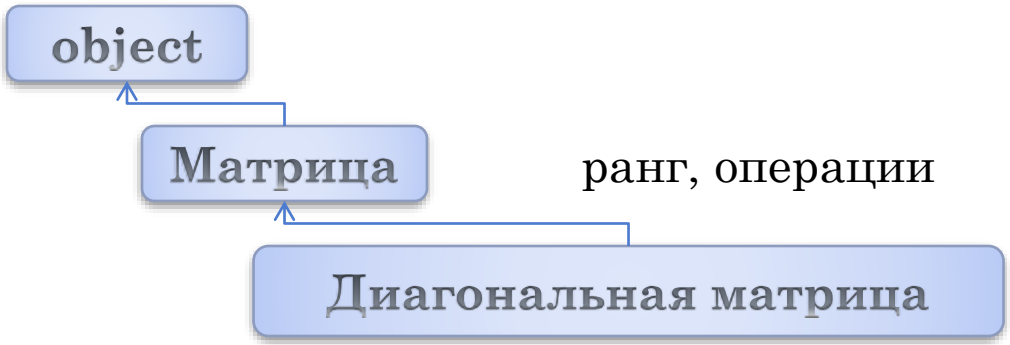
```
class Matrix
{
        public int Rang() { ... };
}

class DiagMatrix : Matrix
{
        public bool isE() { ... };
}

. . . . .

DiagMatrix dm = new DiagMatrix();
int i = dm.Rang();
bool isIdentityMatrix = dm.IsE();
```

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.

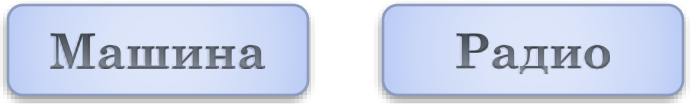
Наследование

- 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 `Object.GetType()`

- Метод `GetType()` возвращает объект типа `System.Type`
- Возвращаемый объект соответствует динамическому типу объекта
- Метод `GetType()` нельзя переопределить
- На этом держится вся безопасность типов 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
  - Сокрытие (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) *

* С C# 9, .NET 5 есть контрвариантность при наследовании

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 { get; set; }
  - 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

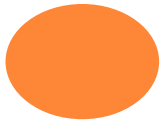
```
class Shape { int Square;}
class Circle : Shape { double Square; }
```

- 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

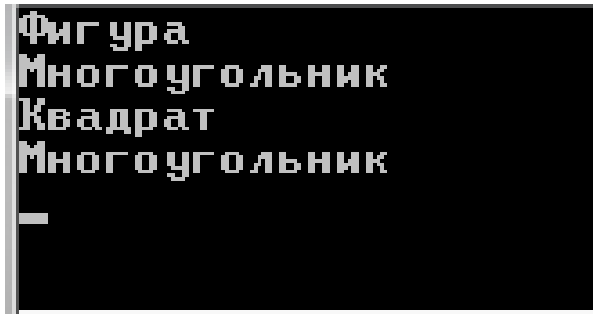
Модификатор new

- 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"; }
    }
}
```



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 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

Инкапсуляция

- Сокрытие реализации, деталей
- Достигается с помощью модификаторов доступа членов типа
  - **public** – член типа виден всем
  - **protected** – член класса виден только внутри самого класса и внутри потомков (для классов)
  - **private** – виден только внутри объявившего его типа *(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 (для классов)
- Модификаторов доступа применяются к типам
  - **public** –тип виден всем
  - **internal** – тип виден только внутри сборки *(по умолчанию)*
  - **file** – тип виден только внутри файла (используется генераторами). **C#11 .NET 7**
- В качестве члена типа может выступать и другой тип. Тогда к нему применимы модификаторы доступа к члену типа

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

Дополнительный сведения о структурах и классах

Отличие класса от структуры

- Большинство синтаксиса и возможностей одинаковые
- Класс- ссылочный тип, структура – тип-значение
- При приравнивании структуры значения всех полей копируется, а при приравнивании 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 C# 12 .NET 8 можно будет переопределять конструктор по умолчанию (пока Release Candidate)
- В отличии 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 оного конструктора Visual Studio создает конструктор по умолчанию
    - public ИмяТипа( ){ }
  - Если задается любой конструктор, конструктор по умолчанию не добавляется

- Struct
  - Всегда имеет неявный public конструктор по умолчанию и его нельзя переопределить (в C# 12 .NET 8 это поменяется)
  - Можно заводить дополнительные конструкторы
    - ComplexStruct myStruct = new ComplexStruct(5,7);
  - Дополнительные конструкторы должны проинициализировать все поля или вызвать конструктор по умолчанию (в C# 12 .NET 8 это поменяется, не проинициализированные поля, проинициализируются автоматически)

READONLY STRUCT И READONLY ЧЛЕНЫ СТРУКТУРЫ

- Структуры могут быть объявлены как readonly

```
readonly struct Incident
    {
        private static int Index = 0;
        private readonly int _id;
        private readonly string _message;
        private readonly DateTime _incidentTime;

        public Incident(string message, DateTime incidentTime)
        {
            _id = ++Index; _message = message; _incidentTime = incidentTime;
        }

        public readonly void Debug()
        {
            Console.WriteLine(_id + " - " + _incidentTime + " - " + _message);
        }

        public readonly override string ToString()
        {
            return _message;
        }
}
```

Компилятор будет контролировать, что все поля в структуре тоже объявлены как readonly

Можно помечать члены как readonly, указывая, что они не меняют состояния (и не в readonly struct)

Readonly члены позволяют избежать копирования объектов при передаче параметров в методы как in

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)

```
partial class A : B, IC {}
partial class A : B, ID {}
partial class A : ID {}
partial class A {}
```

→

```
class A : B, IC, ID
{ суммарный код}
```

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);
}
```

  - <u>Если реализации нет, то все вызовы метода удаляются при компиляции</u>

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 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

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 (допускающим null)

```
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(doudle))
  { … }
```

- 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*/ };
```

  Или

```
if (int.TryParse("55", out int j)) { /*Используем j*/ };
```
- Преобразование различных типов значимые типы: Convert.To*XXX*(object)

  Например:

```
int i = Convert.ToInt32('5');
```
- Преобразование к строке: метод `ToString()`

  Например:

```
int i = 5;
string s = i.ToString();
```

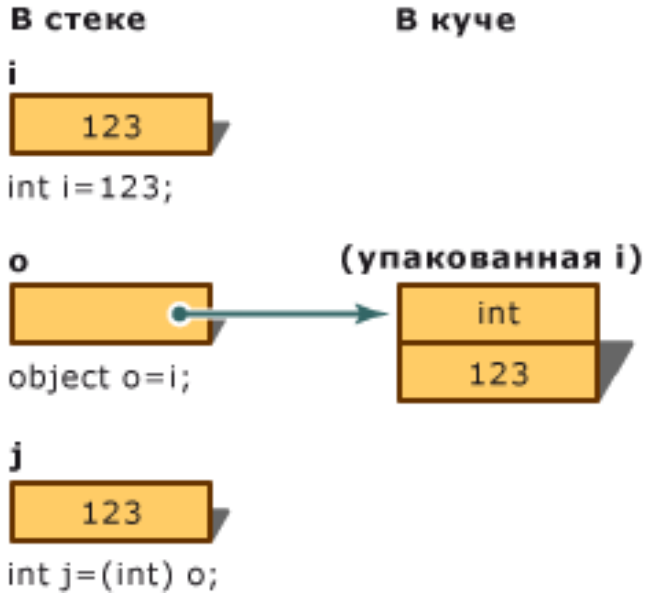
Boxing / Unboxing

- 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

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;
}
```