Введение в классы

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

Введение в ООП. Парадигмы ООП

1. Инкапсуляция
   - *Сокрытие типов*
   - *Сокрытие реализации*
   - *Сокрытие частей программных систем*
2. Наследование
3. Полиморфизм

(использование виртуальных членов, приведение типов, перегрузка операторов и методов)

4. Абстракция

(формирование собирательных понятий)

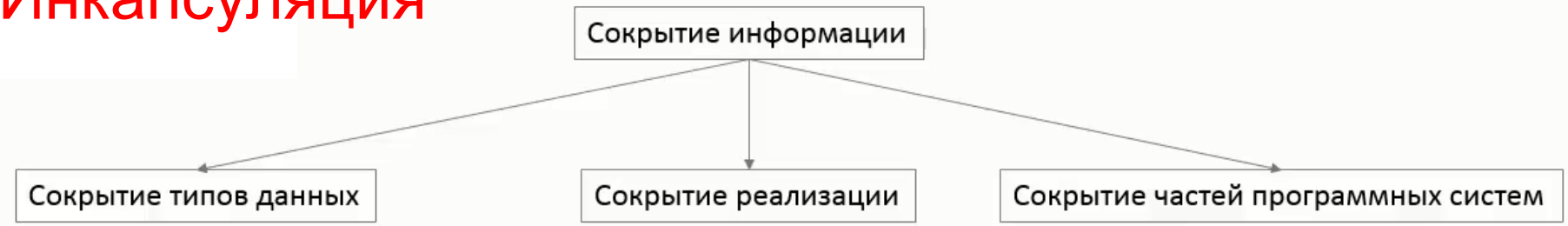
5. Посылка сообщений

(организация информационных потоков между объектами)

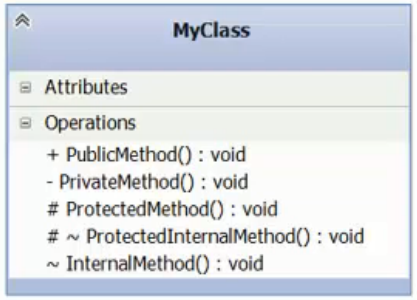
6. Повторное использование

(использование методов, классов, структур, наследования, Библиотек, Фреймворков)

Инкапсуляция



Использование динамических типов `dynamic` и неявно типизированных локальных переменных `var`.



Использование модификаторов доступа
Приведение к базовому типу

Синтаксис объявления класса.

```
[спецификатор] [модификатор] class имя_класса
{
// объявление полей класса
    [спецификатор] [модификатор] тип имя_поля1;
    [спецификатор] [модификатор] тип имя_поля2;
    ….
    [спецификатор] [модификатор] тип имя_поляN;
 // объявление конструкторов
    [спецификатор] [модификатор]  имя_конструктора1 (список параметров)
    {   // тело конструктора
    }
     ….
   [спецификатор] [модификатор]  имя_конструктораN (список параметров)
    {   // тело конструктора
    }

// объявление методов
   [спецификатор] [модификатор] тип имя_метода1(список параметров)
    {   // тело метода
    }
}
```

К членам класса относятся:

- константы,
- поля,
- Конструкторы (типа и экземпляра),
- методы,
- перегруженные операторы,
- операторы преобразования
- свойства,
- события,
- вложенные классы.

Модификаторы доступа языка C#.

- public – данные доступны всем методам во всех сборках
- internal – данные доступны только методам в сборке

- private – данные доступны только методам внутри класса и вложенных в него классам
- protected – данные доступны только методам внутри класса (и вложенным в него классам) или методам из его производных классов
- protected internal – данные доступны только методам вложенного или производного типа класса и любым методам сборки

Класс — это конструкция языка, состоящая из ключевого слова `class`, идентификатора (имени) и тела.

Класс может содержать в своем теле: поля, методы, свойства и события.

Поля определяют состояние, а ***методы*** поведение будущего объекта.

```
class MyClass
{
    public string field;  // Поле

    public void Method()  // Метод
    {
        Console.WriteLine(field);
    }
}
```

Поля класса.

```
class CPerson
    {
        public int Num;          // открытый член класса (поле)
        int ID=102030;           // закрытый член класса
        public string Name;      // открытый член класса
        private string Country = "Беларусь";

        public void Next()       // открытый член класса  (метод)
        {  Num++;
        }
    }
```

Методы класса.

Передача параметров.

Ключевое слово return.

Перегрузка методов.

Передача параметров

```
int i = 0;
int[] myArr = { 0, 1, 2, 4 };

// передаем по значению: i содержит 0, myArr содержит адрес!
MyFunctionByVal1(i, myArr);

Console.WriteLine("i = {0}", i);
Console.WriteLine("MyArr[0] = {0}", myArr[0]);
```

```
static void MyFunctionByVal1(int i, int[] MyArr)
{
  //здесь создается копия этого числа
  i = 100;
  // здесь создается копия адреса
  // обращение к 1-ому элементу массива
  MyArr[0] = 100;
}
```

Передача параметров

```
int i = 0;
int[] myArr = { 0, 1, 2, 4 };

// передаем по значению: i содержит 0, myArr содержит адрес!
MyFunctionByVal(i, myArr);

Console.WriteLine("i = {0}", i);
Console.WriteLine("MyArr[0] = {0}", myArr[0]);
```

```
static void MyFunctionByVal(int i, int[] MyArr)
{
  // здесь создается копия этого числа
  i = 100;
  // здесь создается новый массив
  myArr = new int[] { 3, 2, 1 };
}
```

Использование модификатора out.

Значения **выходных параметров** должны присваиваться вызываемым методом, они передаются по ссылке. Если **выходным параметрам** в вызываемом методе значения **не присвоены**, компилятор сообщит об ошибке

```
static void Main(string[] args)
{
  int ans;
  Add(10, 20, out ans);
  Console.WriteLine("Значение переменной ans: " + ans);
  Console.ReadKey();
}

static void Add(int x, int y, out int ansN)
{
  ansN = x + y; // ansN должно быть присвоено значение
}
```

Использование модификатора ref

Значение первоначально присваивается вызывающим кодом и при желании может быть изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если параметру ref в вызываемом методе значение не присвоено, никакой ошибки компилятор не генерирует

```
static void Main()
  {
   // передаваемые значения по ссылке должны быть проинициализированы
   string str1 = "Первый ";
   string str2 = "Второй ";
   Console.WriteLine("До вызова метода: {0}, {1} ", str1, str2);
   // передача значений str1 и str2  по ссылке
   Metod(ref str1, ref str2);
   Console.WriteLine("После вызова метода: {0}, {1} ", str1, str2);
}

static void Metod(ref string s1, ref string s2)
 {
    string tempStr = s1;
    s1 = s2;
    s2 = tempStr;
 }
```

Создание методов с переменным количеством аргументов.

```
static void Main(string[] args)
{
  Console.WriteLine(Metod(1, 2, 3, 4, 5));
  Console.WriteLine(Metod(1, 2, 3, 4, 5,6,7,8,9,10));
  Console.WriteLine(Metod(new int []{5, 6, 7, 8}));
}
// метод с переменным количеством аргументов
// params - должен быть последним и только одним
public static int  Metod(params int [] mas)
{
   return mas.Length;
}
```

Передача параметров

```
int i = 0;
int[] myArr = { 0, 1, 2, 4 };

// передаем по ссылке
MyFunByRef(ref i, ref myArr);

Console.WriteLine("i = {0}", i);
Console.WriteLine("MyArr[0] = {0}", myArr[0]);
```

```
static void MyFunByRef(ref int i, ref int[] MyArr)
{

  i = 100;

  myArr = new int[] { 3, 2, 1 };
}
```

Перегрузка методов.

```
// перегрузка метода (два параметра типа int)
public static int Sum(int a, int b)
{
  return a + b;
}
// перегрузка метода (три параметра типа int)
public static int Sum(int a, int b, int c)
{
return a + b + c;
}
// перегрузка метода (два параметра типа double)
public static double Sum(double a, double b)
{
  return a + b;
}
```

допускается!

допускается!

допускается!

перегрузка метода НЕ допускается!

(два параметра типа double, но тип возврата string)

```
//public static string Sum(double a, double b)
//{
//    return (a + b).ToString();
//}
```

Конструкторы

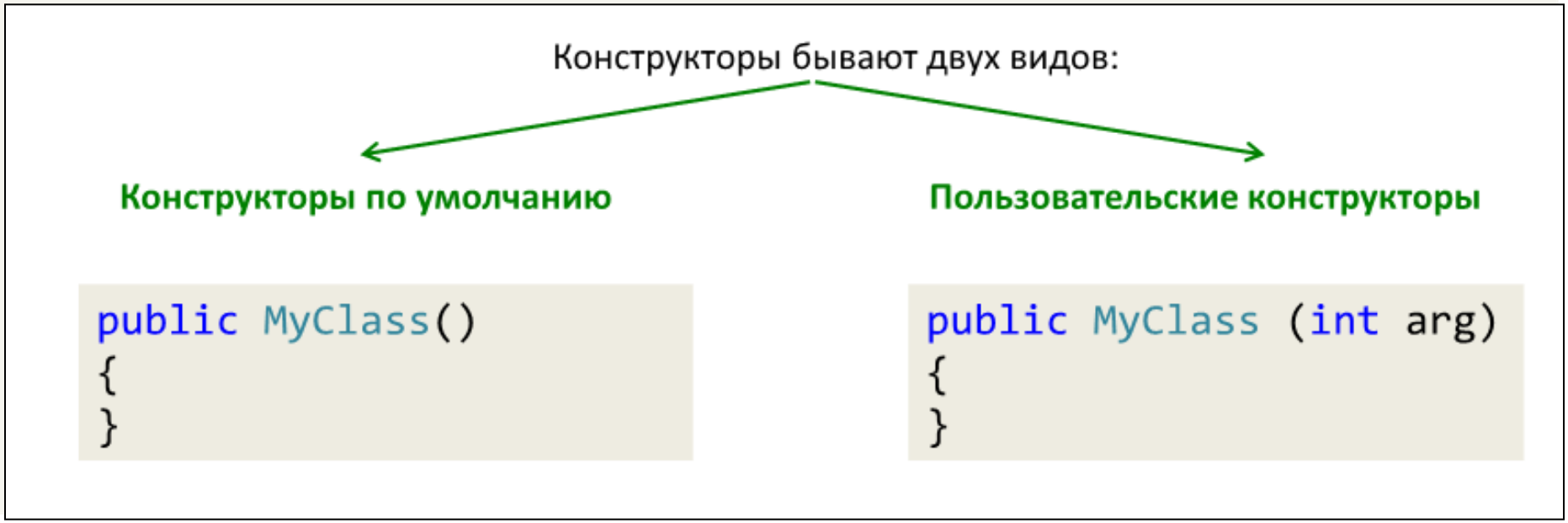
Понятие конструктора.

Параметризованный конструктор.

Перегруженные конструкторы.

Статические конструкторы.

Конструктор класса - специальный метод, который предназначен для инициализации полей класса и вызывается при построении экземпляра класса.



Задача конструктора по умолчанию – инициализация полей значениями по умолчанию.

Задача пользовательского конструктора – инициализация полей предопределенными пользователем значениями.

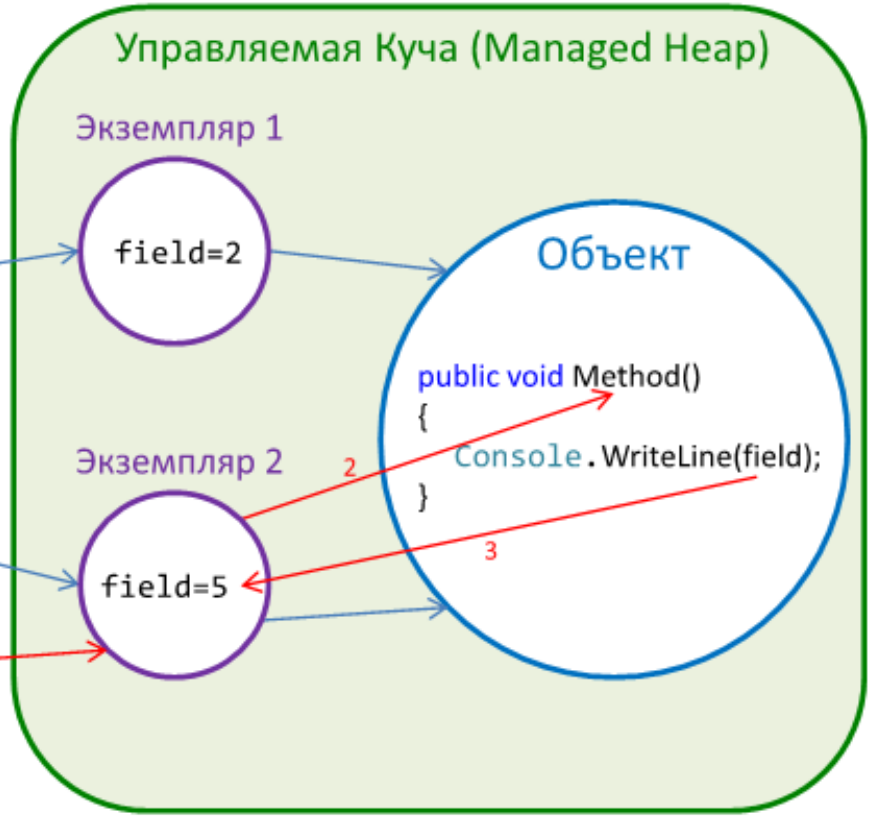
Объекты содержат в себе статические поля и все методы.

Экземпляры содержат нестатические поля.

```
MyClass instance1 = new MyClass();
MyClass instance2 = new MyClass();

instance1.field=2;
instance2.field=5;

instance1.Method();
instance2.Method();
```



Сцепление конструкторов или цепочка конструкторов

```
// Конструктор без параметров
        public Car() : this("Нет водителя")
        {         }
// Конструктор с одним параметром
        public Car(string driverName) : this(driverName, 0)
        {         }
// Конструктор с параметрами
        public Car(string driverName, int speed)
        {   this.driverName = driverName;
            this.currSpeed = speed;
        }
```

```
Console.WriteLine("Конструктор по умолчанию");
Car myCar = new Car();
myCar.PrintState();
// Вывод – Нет водителя, скорость=0

Console.WriteLine("Конструктор с параметрами");
myCar = new Car("Рубенс Барикелло",50);
myCar.PrintState();
// Вывод – Рубенс Барикелло, скорость=50
```

Конструктор копии

```
class Person
    {   public int Age;
        public string Name;
        // Конструтор экземпляра
        public Person(string name, int age)
        {   Name = name;  Age = age;
        }
        // конструктор копии
        public Person(Person prevPerson)
        {   Name = prevPerson.Name;  Age = prevPerson.Age;
        }
}
```

```
// Создание объекта класса Person
Person person1 = new Person("Иван", 40);
// Создание копии объекта person1
Person person2 = new Person(person1);
// Изменение значений
person1.Age = 39;
person2.Age = 41; person2.Name = "Андрей";
```

Свойства

Свойство это конструкция языка C#, которая заменяет собой использование обычных методов доступа.

```
int field;

public int Property
{
    get
    {
        return field;
    }

    set
    {
        field = value;
    }
}
```

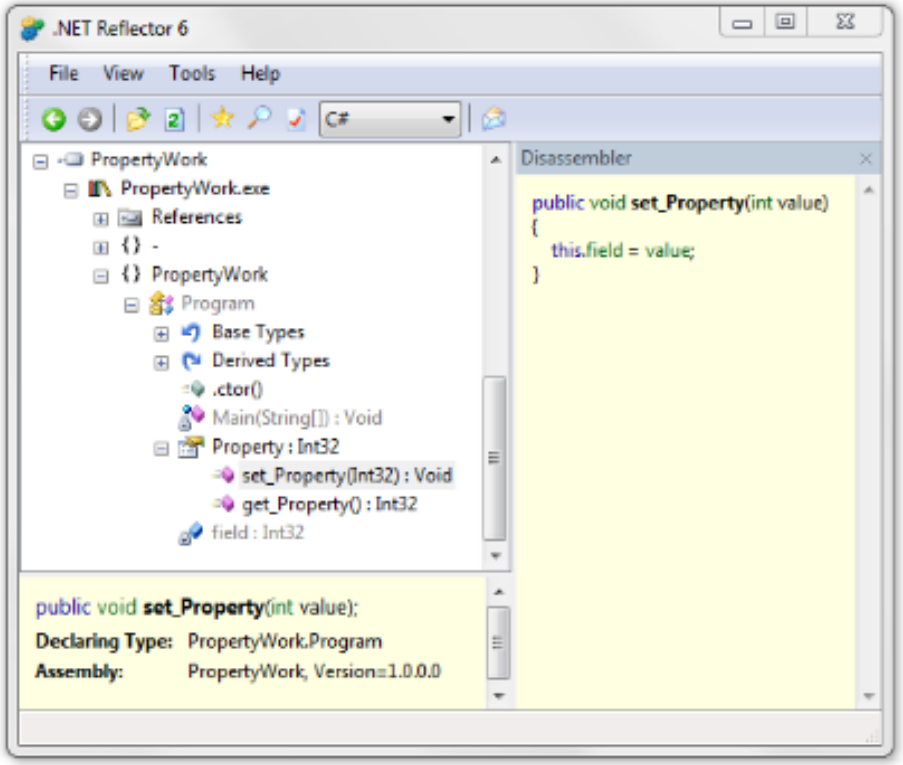
Работа со свойством экземпляра напоминает работу с полями экземпляра.

Свойство состоит из имени, типа и тела. В теле задаются методы доступа, через использование ключевых слов `set` и `get`.

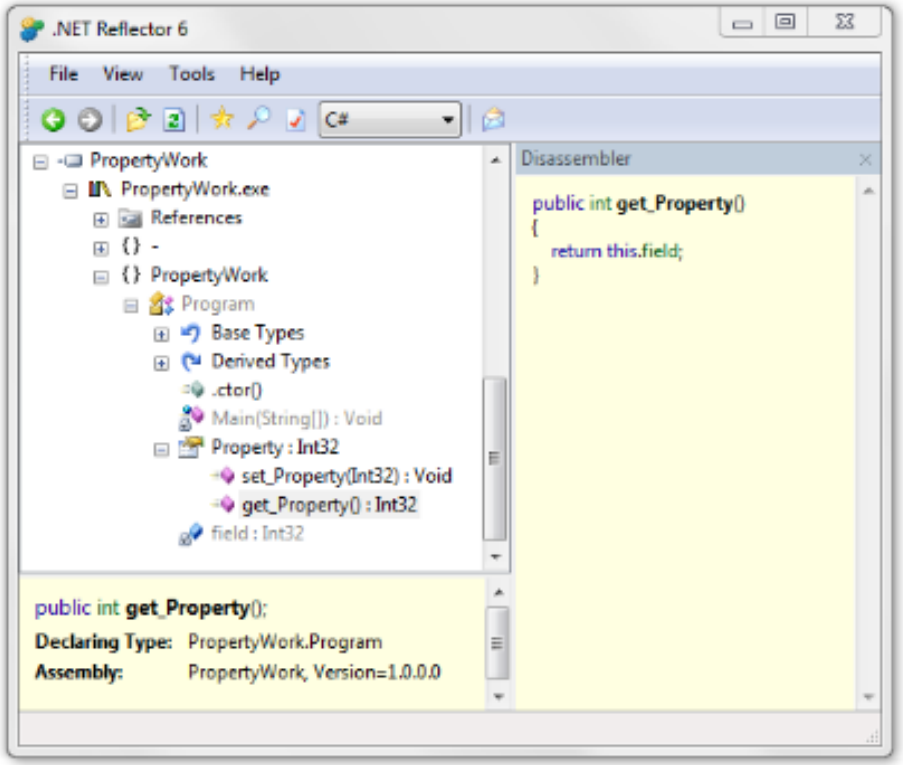
Метод `set` автоматически срабатывает тогда, когда свойству пытаются присвоить значение. Это значение представлено ключевым словом `value`.

Метод `get` автоматически срабатывает тогда, когда мы пытаемся получить значение.

Анализ кода реализации свойств с использованием программы .NET Reflector.



Метод доступа set



Метод доступа get

Классы и его статические члены

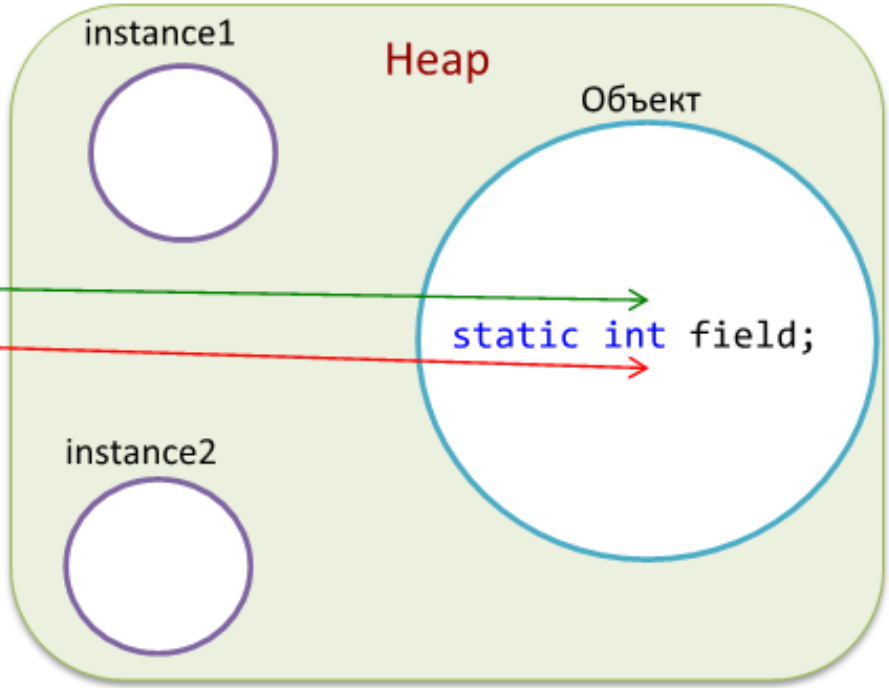
Статическая переменная - это общая переменная для всех экземпляров класса, которая хранится в объекте.

Объекты содержат в себе статические поля и методы.

```
static void Main()
{
    MyClass instance1 = new MyClass();
    MyClass instance2 = new MyClass();

    MyClass.field = 2;
    MyClass.field = 5;
}
```

```
class MyClass
{
    public static int field;
}
```



Классы и его статические члены

```
class SomeClass
{
 int number;
 static int count;

public static void Continue()
 { count++;  }

 public void CountUp()
 {
    number++;
    count++;
    Continue();
 }
}
```

Статические методы

```
class SomeClass
    {
        int number;
        static int count;
        public static void Continue()
        {
            count++;
            number++; // СТЕ !!!
        }
    }
```

Статические методы не могут обращаться к нестатическим полям класса.

Статические классы

Существуют классы, не предназначенные для создания экземпляров, В сущности, такие классы существуют лишь для группировки логически связанных членов.

В C# такие классы определяются с ключевым словом **static**. Его разрешается применять только к классам, но не к структурам (значимым типам).

```
static class SomeStaticClass
{
    static int count;
    public static void Show() { }
    public static void Continue() { }
}
```

Особенности статических классов

Компилятор налагает на статический класс ряд ограничений.

- В классе можно определять только статические члены (поля, методы, свойства и события). Любые экземплярные члены вызовут ошибку компиляции.
- Класс нельзя использовать в качестве поля, параметра метода или локальной переменной, поскольку это подразумевает существование переменной, ссылающейся на экземпляр, что запрещено. Обнаружив подобное обращение со статическим классом, компилятор вернет сообщение об ошибке.
- Класс должен быть прямым потомком System.Object - наследование любому другому базовому классу лишено смысла, поскольку наследование применимо только к объектам, а создать экземпляр статического класса невозможно.
- Класс не должен реализовывать интерфейсов, поскольку методы интерфейса можно вызывать только через экземпляры класса.

Статический класс – это контейнер, который содержит в себе только статические члены.

Доступ к членам статического класса осуществляется на **Классе-Объекте**

```
static class  StaticClass
{
    public static int item;
}
```

```
static void Main()
{
    StaticClass.item = 10;
}
```

Если класс содержит статические поля, должен быть предоставлен статический конструктор, который инициализирует эти поля при загрузке класса.

Классы и статические классы могут иметь статические конструкторы.

```
class MyClass
{
    public static int field;

    static MyClass()
    {
        field = 10;
    }
}
```

Статический конструктор всегда отрабатывается первым (перед первым обращением к классу).

Свойства статического конструктора

- Статический конструктор не имеет модификаторов доступа и не принимает параметров.
- Статический конструктор вызывается автоматически для инициализации класса перед созданием первого экземпляра или ссылкой на какие-либо статические члены.
- Статический конструктор нельзя вызывать напрямую.
- Пользователь не управляет тем, когда статический конструктор выполняется в программе.

Константы

Константа (const) - это идентификатор, отмечающий поле, значение которого никогда не меняется. При определении идентификатора константы компилятор должен получить его значение во время компиляции. Затем компилятор сохраняет значение константы в метаданных модуля.

константы считаются статическими, а не экземплярными членами.

readonly - запись в поле разрешается только из кода конструктора

Частичные типы (partial types).

```
// Объявление частичного класса
   partial class Person
   {
       public String Phone;
       public String Email;
   }

   class Program
   {   static void Main(string[] args)
       {   Person person = new Person();
           person.Num = 1;
           person.Name = "Иван";
           person.Email = "kin@tut.by";
           person.Phone = "+375298625532";
       }
   }
   // Объявление частичного класса
   partial class Person
   {
       public Int16 Num;
       public String Name;
   }
```

Дополнительно

SOLID (принципы дизайна классов в объектно-ориентированном проектировании)

- Принцип единственной ответственности (Single responsibility)
- Принцип открытости/закрытости (Open-closed)
- Принцип подстановки Барбары Лисков (Liskov substitution)
- Принцип разделения интерфейса (Interface segregation)
- Принцип инверсии зависимостей (Dependency Invertion)

Принцип единственной ответственности гласит — *«На каждый объект должна быть возложена одна единственная обязанность»* — конкретный класс должен решать конкретную задачу — ни больше, ни меньше.

DRY: Don’t Repeat Yourself

Достигается за счет того, что

- Отсутствием copy-paste;
- Повторного использования кода.

https://habrahabr.ru/post/208442/

Требования к наименованиям

Стиль **Pascal case** применяется **к методам и свойствам** – каждое слово в имени метода начинается с верхнего регистра без символа нижнего подчеркивания. Глаголы.

Стиль **Camel case** (**для локальных переменных и полей типа**) – первое слово в нижнем регистре, а все остальные начинаются с буквы верхнего регистра. Существительные.

Стиль **Upper case** (**для имен констант**) все слова содержат буквы верхнего регистра.

Стиль **Hungarian case** – в начале в нижнем регистре сокращенно тип идентификатора, а далее все слова начинаются с верхнего регистра. Только для интерфейсов и в generic тип.

Группировка в регионы:

```
#region Fiels and Constants

private int id;

private string autorName;

#endregion
```

```
#region Constructor
0 references
public Book(string Author)
{
    this.AuthorName = Author;

}
#endregion
```

```
#region Property

3 references
public int ID {
    get
    {
        return id;
    }
    private set { }
}

1 reference
public string AuthorName { get; set; }

#endregion
```

Документирование приложений. XML комментарии

Создание документированного компонента включает несколько шагов:

1. Создание компонента
2. Создание XML файла
3. Тестирование компонента в сценарии развертывания