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)

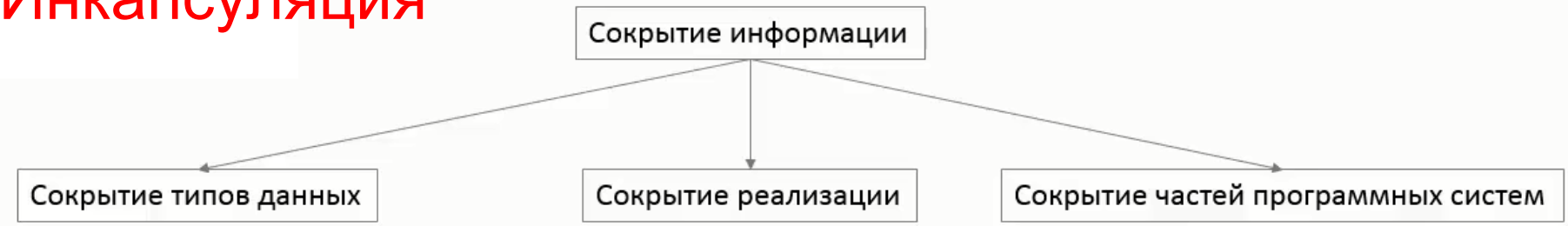
5. Посылка сообщений

(организация информационных потоков между объектами)

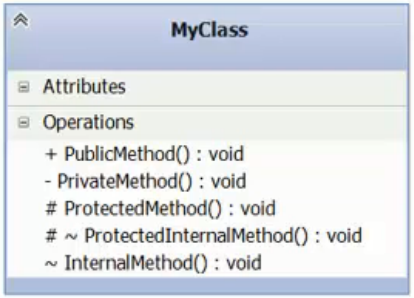
6. Повторное использование

(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
    }
}
```

Модификаторы доступа типа

- ***Открытый*** (public) тип доступен любому коду любой сборки.
- ***Внутренний*** (internal) тип доступен только в той сборке, где он определен.

Внимание! По умолчанию компилятор C# делает тип внутренним (с более ограниченной видимостью)

```
// Открытый тип доступен из любой сборки
public class ThisIsAPublicType { ... }
// Внутренний тип доступен только из собственной сборки
internal class ThisIsAnInternalType { ... }
// Это внутренний тип, так как модификатор доступа не указан явно
class ThisIsAlsoAnInternalType { ... }
```

Доступ к членам типа (класса)

- private – доступны только методам в определяющем типе и вложенных в него типах.
- protected – доступны только методам в определяющем типе (и вложенным в него типах) или в одном из его производных типов независимо от сборки
- internal – доступны только методам в определяющей сборке
- protected internal – доступны только методам вложенного типа, производного типа (независимо от сборки) и любым методам определяющей сборки
- public – данные доступны всем методам во всех сборках

К членам класса относятся:

- константы,
- поля,
- конструкторы (типа и экземпляра),
- методы,
- перегруженные операторы,
- операторы преобразования
- свойства,
- индексаторы,
- события,
- типы (вложенные классы)

Модификаторы доступа языка C#.

```
public sealed class SomeType
{       // Вложенный класс
        private class SomeNestedType { }
        // Константа, неизменяемое и статическое изменяемое поле
        // Constant, readonly, and static read/write field
        private const Int32 c_SomeConstant = 1 ;
        private readonly String m_SomeReadOnlyField = "2";
        private static Int32 s_SomeReadWriteField = 3;
        // Конструктор типа
        static SomeType() { }
        // Конструкторы экземпляров
        public SomeType(Int32 x) { }
        public SomeType() { }
        // Экземплярный и статический методы
        private String InstanceMethod() { return null; }
        public static void Main() {}
        // свойство
        public Int32 SomeProp
        {   get { return 0; }  set { }   }
        // индексатор
        public Int32 this[String s]
        {     get { return 0; }  set { }     }
        // Экземплярное событие
        public event EventHandler SomeEvent;
 }
```

Константы

Константа (const) - это идентификатор, отмечающий поле, значение которого никогда не меняется.

Особенности:

- Значение константы определяется во время компиляции т.е. явно задается.
- Компилятор сохраняет значение константы в метаданных модуля
- Значение константы можно определять для типов, которые компилятор считает примитивными (особенность string)
- константы считаются статическими, а не экземплярными членами

Встретив в исходном коде имя константы, компилятор просматривает метаданные модуля, в котором она определена, извлекает значение константы и внедряет его в генерируемый IL-код. Поскольку значения констант внедряются прямо в код, то в период выполнения память для них не выделяется.

Константы

```
public class SomeClass
{
 private const int Max_Size = 1000;
 public const string Directory = "D:\\Files";
 private const SomeType Empty = null;
 private const SomeType NoEmpty = new SomeType();  // CTE
}
```

Error 1 ‘Example.SomweClass.NoEmpty' is of type 'Example.SomeType'. A const field of a reference type other than string can only be initialized with null

```
static void Main(string[] args)
{
   // обращение к константе
   Console.WriteLine(SomeClass.Directory);
   Console.ReadKey();
}
```

Поля

Поле (field) - это член данных, который хранит экземпляр значимого типа или ссылку на ссылочный тип

Особенности:

- имеются поля типов (статические) и поля экземпляров (нестатические).
- Память для статических полей выделяется при первом обращении к типу, для нестатических полей - при создании экземпляра типа
- Значением поля может быть любой тип
- Имеются поля, предназначенные для чтения и записи (изменяемые поля), а также только для чтения (неизменяемые) поля, помеченные readonly. Неизменяемые поля могут быть проинициализированы только при объявлении и исполнении конструктора.

Поля

```
public class SomeClass
{
    private readonly int Max_Size = 1000;
    private readonly string Directory = "D:\\Files";
    private readonly decimal Round;
    private readonly SomeType Empty = null;
    private readonly SomeType NoEmpty = new SomeType
    public SomweClass()
    {
      Round = 0.001m;
    }
    public void SomeMetod() // CTE
    {
      Round = 0.001m;
    }
}
```

Error 1 A readonly field cannot be assigned to (except in a constructor or a variable initializer)

Конструкторы.

Конструкторы экземпляров.

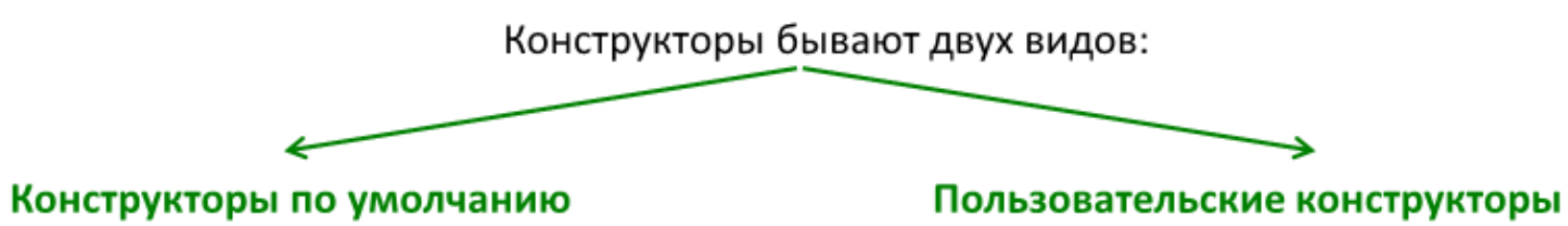
Конструкторы типов (статические конструкторы).

Конструкторы экземпляров

Конструктор (.ctor - constructor) - это специальный метод, позволяющий корректно инициализировать новый экземпляр типа

Особенности:

- Поля не устанавливаемые явно конструктором, гарантированно инициализируются значениями по умолчанию (значимые типы 0, ссылочные типы - null).
- В классе где явно не задан конструктор, компилятор создает конструктор по умолчанию (без параметров)
- Конструкторы не наследуются
- К конструкторам нельзя применить ключевые слова virtual, abstract, virtual, new, sealed.
- Для абстрактных классов компилятор создает конструктор с модификатором protected
- В статических классах компилятор не создает конструктор по умолчанию.




```
public MyClass()
{
}
```

```
public MyClass (int arg)
{
}
```

Задача конструктора по умолчанию – инициализация полей значениями по умолчанию.

Задача пользовательского конструктора – инициализация полей предопределенными пользователем значениями.

Конструкторы экземпляров

```
public class SomeType
{ }
 // вид конструктора по умолчанию
public class  SomeType
{
    public SomeType():base() { }
    public SomeType(int num)
    { this.num = num; }
}
```

```
static void Main(string[] args)
{
  // создание экземпляра типа
   SomeType obj1 = new SomeType();
   SomeType obj2 = new SomeType(10);
}
```

Конструкторы экземпляров

```
public class  SomeType
{
  private int id=5;
  private string sourse="data.xml";
  private double round;
  public SomeType() {}
  public SomeType(string sourse)
    {this.sourse=sourse; }
}
```

```
SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  ldc.i4.5
IL_0002:  stfld     UserQuery+SomeType.id
IL_0007:  ldarg.0
IL_0008:  ldstr     "data.xml"
IL_000D:  stfld     UserQuery+SomeType.sourse
IL_0012:  ldarg.0
IL_0013:  call      System.Object..ctor
IL_0018:  nop
IL_0019:  nop
IL_001A:  ret
```

```
SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  ldc.i4.5
IL_0002:  stfld     UserQuery+SomeType.id
IL_0007:  ldarg.0
IL_0008:  ldstr     "data.xml"
IL_000D:  stfld     UserQuery+SomeType.sourse
IL_0012:  ldarg.0
IL_0013:  call      System.Object..ctor
IL_0018:  nop
IL_0019:  nop
IL_001A:  ldarg.0
IL_001B:  ldarg.1
IL_001C:  stfld      UserQuery+SomeType.sourse
IL_0021:  ret
```

Конструкторы экземпляров

```
public class  SomeType
{
  private int id;
  private string sourse;
  private double round;
  public SomeType()
   {
     sourse="data.xml";
     id=5;
   }
  public SomeType(string sourse)
   {
     this.sourse=sourse;
     id=5;
   }
}
```

```
SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  call      System.Object..ctor
IL_0006:  nop
IL_0007:  nop
IL_0008:  ldarg.0
IL_0009:  ldstr     "data.xml"
IL_000E:  stfld     UserQuery+SomeType.sourse
IL_0013:  ldarg.0
IL_0014:  ldc.i4.5
IL_0015:  stfld     UserQuery+SomeType.id
IL_001A:  ret

SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  call      System.Object..ctor
IL_0006:  nop
IL_0007:  nop
IL_0008:  ldarg.0
IL_0009:  ldarg.1
IL_000A:  stfld     UserQuery+SomeType.sourse
IL_000F:  ldarg.0
IL_0010:  ldc.i4.5
IL_0011:  stfld     UserQuery+SomeType.id
IL_0016:  ret
```

Конструкторы экземпляров

```
public class  SomeType
{
  private int id;
  private string sourse;
  private double round;
  public SomeType():this("data.xml")
    { }
  public SomeType(string sourse)
    {
       this.sourse=sourse;
       id=5;
    }
}
```

```
SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  ldstr      "data.xml"
IL_0006:  call       UserQuery+SomeType..ctor
IL_000B:  nop
IL_000C:  nop
IL_000D:  ret

SomeType..ctor:
IL_0000:  ldarg.0
IL_0001:  call       System.Object..ctor
IL_0006:  nop
IL_0007:  nop
IL_0008:  ldarg.0
IL_0009:  ldarg.1
IL_000A:  stfld      UserQuery+SomeType.sourse
IL_000F:  ldarg.0
IL_0010:  ldc.i4.5
IL_0011:  stfld      UserQuery+SomeType.id
IL_0016:  ret
```

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";
```

Методы класса.

Передача параметров.

Операторы ref, out, params.

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 **должно быть установлено** перед вызовом и может быть изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если параметру ref в вызываемом методе значение не присвоено, то ошибку 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;
 }
```

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 };
}
```

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}));
}
public static int  Metod(params int [] mas)
{
   return mas.Length;
}
```

Особенности

- **params** - в списке параметров может быть использовать только один раз
- **params** - при использовании должен быть последним

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();
//}
```

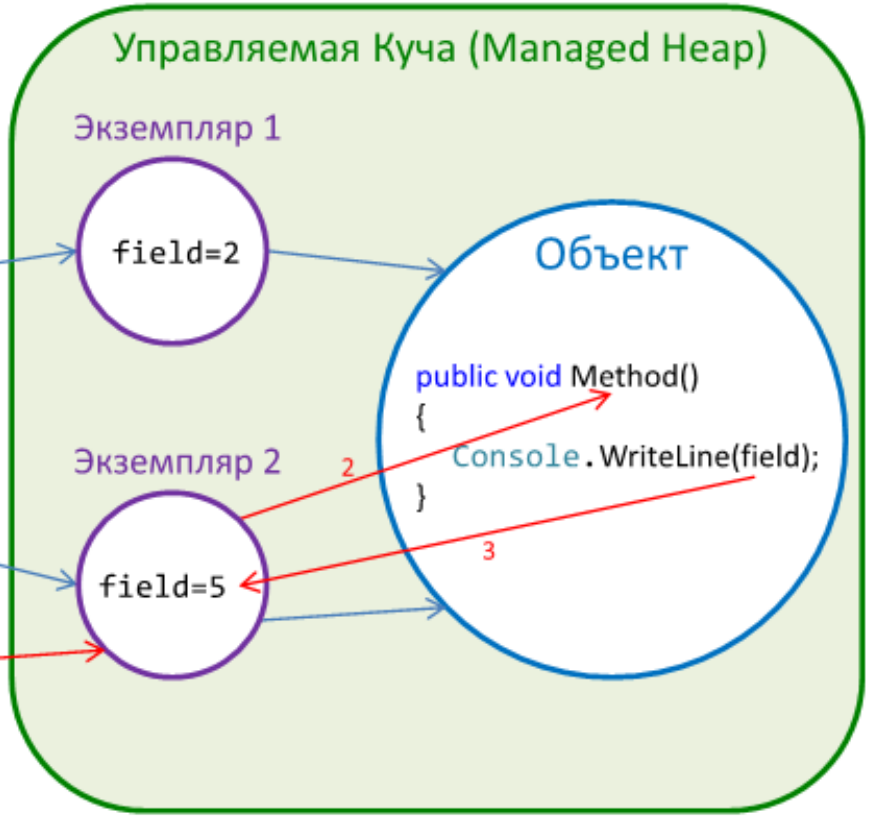
Объект и экземпляр

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();
```



Классы и его статические члены

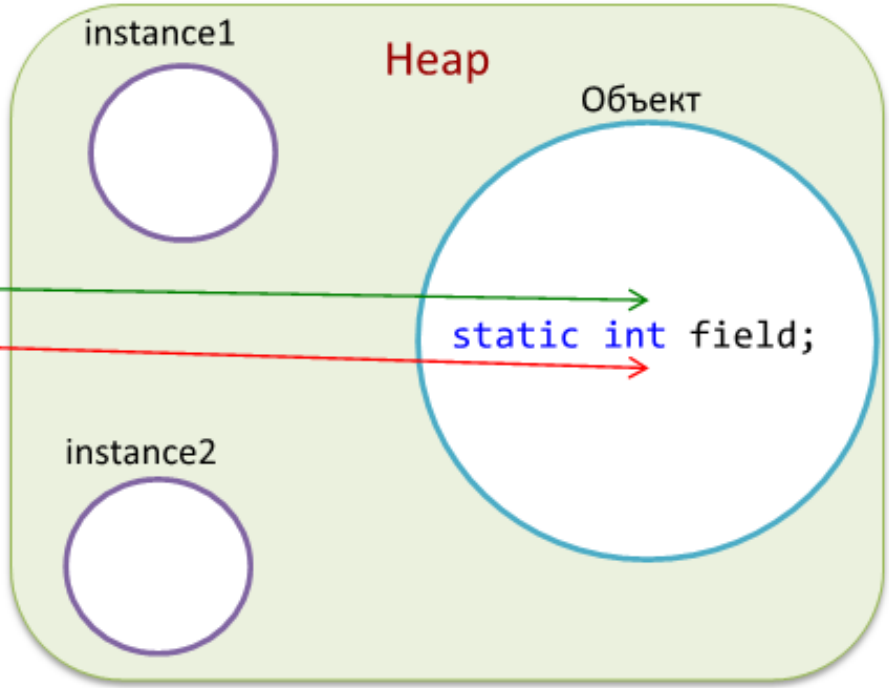
Статическая переменная - это общая переменная для всех экземпляров класса, которая хранится в объекте.

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;
}
```



Статические классы

Существуют классы, не предназначенные для создания экземпляров. Классы данного типа используются для **группировки логически связанных членов**.

Объект статического класса создать невозможно.

```
public static class Math
{
    public const double E = 2.71828;
    public const double PI = 3.14159;
    public static decimal Abs(decimal value){}
    public static double Cos(double d){}
    public double Cos(double d){}          // CTE 1
    public static int this[int index]{}    // CTE 2
}
```

Error 1 cannot declare instance members in a static class
Error 2 'Math.this[int]': cannot declare indexers in a static class

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.

Конструкторы типов (статические конструкторы)

Конструктор типа используется для установки первоначального состояния типа (инициализации статических полей).

Особенности:

- По умолчанию у типа не определено конструктора
- У типа может быть только один конструктор
- У конструктора типа не может быть параметров
- У конструкторов типа не должно быть модификаторов доступа (в C# конструктор типа всегда закрытый)
- Конструктор типа невозможно вызвать напрямую, разработчик не имеет возможности управлять тем, когда будет вызван конструктор типа
- Конструктор типа вызывается автоматически только один раз перед созданием первого экземпляра типа или ссылкой на статические члены т.е. перед первым обращением к типу.

Конструкторы типов (статические конструкторы)

```
internal static class SomeType
{
    private int number; // CTE
    private static int capacity;
    static SomeType()
    {
        capacity = 1000;
    }
}
```

Error 1 'SomeType.number': cannot declare instance members in a static class

Конструкторы типов (статические конструкторы)

```
internal class SomeType
{
    private int number;
    private static int capacity;
    static SomeType()
    {
        capacity = 1000;
        number++;                  // CTE
    }
    public SomeType()
    {
        capacity = 1000;
        number++;
    }
}
```

Error 1 An object reference is required for the non-static field, method, or property 'ConsoleApplication2.SomeType.number

Конструкторы типов (статические конструкторы)

```
SomeType..cctor:
IL_0000:  ldc.i4.5
IL_0001:  stsfld      UserQuery+SomeType.capacity
IL_0006:  ret
```

```
internal static class SomeType
{
    private static int capacity=1000;
}
```

При компоновке кода, в котором имеется инициализация статического поля при объявлении, компилятор автоматически создает конструктор типа с инициализацией поля.

```
internal static class SomeType
{
    private static int capacity;
    static SomeType()
    {
        capacity = 1000;
    }
}
```

Непригодный тип

```
static void Main(string[] args)
{

    SomeType.Method();

    Console.ReadKey();
}
```

TypeInitializationException was unhandled

An unhandled exception of type 'System.TypeInitializationException' occurred in ConsoleApplication2.exe

Если в конструкторе типа генерируется необработанное исключение, то при попытке обращении к любому члену типа возникает исключение и тип непригодный

```
internal static class SomeType
{
    private static int capacity;
    static SomeType()
    {   capacity = 1000;
        throw new Exception();  // RTE
    }
    public static void Method()
    {
        capacity *= 2;
    }
}
```

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;
   }
```

Частичные методы

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