

Современные платформы программирования. Часть 1. Технология .NET

Дмитрий Андреевич Сурков
Иван Владимирович Шимко

© Полное или частичное копирование материалов без
письменного разрешения авторов запрещено.

Источники

- Конспект лекций с примерами программ
- Д.А. Сурков, К.А. Сурков, Ю.М. Четырько — Модели утилизации динамической памяти
- Джейфри Рихтер — «CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#»
- <https://docs.microsoft.com/dotnet/>
- Борис Бабаян — Защищённые информационные системы

Направление развития платформ программирования

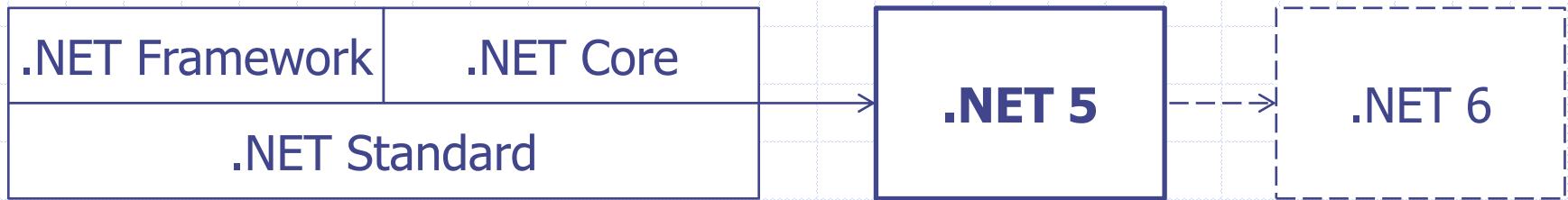
- Повышение уровня абстракции:
- Концепции языков высокого уровня на уровне исполняющей системы
- Автоматическое управление памятью
- Промежуточный язык и динамическая компиляция
- Информация о типах в исполняемых модулях
- Безопасность и верифицируемость кода

Экскурс в историю

- Работы по теории программирования академика А.П. Ершова. Внутренний (промежуточный) язык. Лексикон.
- Вычислительные системы Эльбрус
- Система программирования Оберон
- Платформа Java
- Платформа .NET
- Что дальше?

Платформа .NET

- **.NET Framework** – первая реализация для Windows.
- **.NET Core** – кроссплатформенная реализация с открытым исходным кодом (Windows, Linux, macOS).
- **.NET Standard** – спецификация API, позволяющая создавать библиотеки для использования и в .NET Framework, и в .NET Core (согласно таблице совместимости версий).
- **.NET 5+** – единая платформа, замещающая .NET Framework и .NET Core.



Сравнительная характеристика технологий .NET и Java

- Байт-код / промежуточный язык (IL):
 - Java – интерпретация
 - .NET – динамическая компиляция
- Структурные типы данных
 - Java – только в динамической памяти
 - .NET – на стеке и в динамической памяти
- Многомерные массивы
 - Java – «ступенчатые» массивы
 - .NET – «прямоугольные» и «ступенчатые» массивы
- Методы объектов
 - Java – все виртуальные
 - .NET – виртуальные или не виртуальные
- Возврат результатов из подпрограмм
 - Java – как только одно значение функции
 - .NET – как значение функции и в ref- и out-параметрах

Принципы .NET

- Переносимость программ между аппаратурой и ОС благодаря промежуточному коду (IL) и динамической компиляции
- Модульное программирование на основе сборок (assembly) с версионностью и полной информацией о типах
- Автоматическое управление памятью на основе сборки мусора
- Объектная модель с ссылочными и «скалярными» типами в основе платформы
- Открытость и расширяемость, поддержка многих языков

Минимальная программа для платформы .NET

```
using System;

namespace HelloWorldDisassembled
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello!!!");
            Console.ReadLine();
        }
    }
}
```

Восстановление исходного кода из промежуточного (де-компиляция)

- Утилита ILSpy

The screenshot shows the ILSpy application window. On the left, the 'Assemblies' tree view is expanded to show the contents of 'mscorlib (4.0.0.0, .NETFramework, v4.0)'. The 'System.Collections.Generic' assembly is selected. On the right, the decompiled source code for the `List<T>` class is displayed in a code editor. The code is color-coded, with comments in green and keywords in blue. The class implements several interfaces like `IList<T>`, `ICollection<T>`, etc.

```
// System.Collections.Generic.List<T>
using ...
[Serializable]
[DebuggerTypeProxy(typeof(Mscorlib_CollectionDebugView<>))]
[DebuggerDisplay("Count = {Count}")]
[__DynamicallyInvokable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
    [Serializable]
    internal class SynchronizedList : IList<T>, ICollection<T>, IEnumerable<T>, IDisposable, IEnumerator
    ...

    [Serializable]
    [__DynamicallyInvokable]
    public struct Enumerator : IEnumerator<T>, IDisposable, IEnumerator
    ...

    private const int _defaultCapacity = 4;

    private T[] _items;

    private int _size;

    private int _version;

    [NonSerialized]
    private object _syncRoot;

    private static readonly T[] _emptyArray = new T[0];

    [__DynamicallyInvokable]
    public int Capacity
    ...
}
```

Деассемблированная программа для платформы .NET

```
.namespace HelloWorldDisassembled
{
    .class private auto ansi beforefieldinit HelloWorldDisassembled.Program
        extends [mscorlib]System.Object
    {
        .method private hidebysig static void Main(string[] args) cil managed
        {
            .maxstack 8
            .entrypoint

            // Console.WriteLine("Hello!!!");
            IL_0000: ldstr "Hello!!!"
            IL_0005: call void [mscorlib]System.Console::WriteLine(string)
            // Console.ReadLine();
            IL_000a: call string [mscorlib]System.Console::ReadLine()
            IL_000f: pop
            // (no C# code)
            IL_0010: ret
        } // end of method Program::Main

        .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
        {
            .maxstack 8

            IL_0000: ldarg.0
            IL_0001: call instance void [mscorlib]System.Object::.ctor()
            IL_0006: ret
        } // end of method Program::..ctor
    } // end of class HelloWorldDisassembled.Program
}
```

Деассемблированная программа для режима Debug

```
.namespace HelloWorldDisassembled
{
    .class private auto ansi beforefieldinit HelloWorldDisassembled.Program
        extends [mscorlib]System.Object
    {
        .method private hidebysig static void Main(string[] args) cil managed
        {
            .maxstack 8
            .entrypoint

            // (no C# code)
            IL_0000: nop
            // Console.WriteLine("Hello!!!");
            IL_0001: ldstr "Hello!!!"
            IL_0006: call void [mscorlib]System.Console::WriteLine(string)
            // (no C# code)
            IL_000b: nop
            // Console.ReadLine();
            IL_000c: call string [mscorlib]System.Console::ReadLine()
            IL_0011: pop
            // (no C# code)
            IL_0012: ret
        } // end of method Program::Main

    } // end of class HelloWorldDisassembled.Program
}
```

Виды программного кода в платформе .NET: managed/unmanaged, safe/unsafe

- По уровню абстракции программный код может быть:
 - **managed** – управляемый, с поддержкой сборки мусора; транслируется в промежуточный код **IL**
 - **unmanaged** – неуправляемый, без поддержки сборки мусора; транслируется в **машинный код**
- По уровню безопасности программный код может быть:
 - **safe** – безопасный, адресная арифметика запрещена
 - **unsafe** – небезопасный, адресная арифметика разрешена
- Разрешённые комбинации:
 - **managed – safe**
 - **managed – unsafe**
 - **unmanaged – unsafe**
- Разрешено сочетание и взаимодействие всех видов кода

Небезопасный код (unsafe). Адресная арифметика разрешена

```
public static void UnsafeToLower(char[] array)
{
    unsafe
    {
        fixed (char* a = &array[0])
        {
            char* p = a;
            while (*p != '\0')
            {
                *p = char.ToLower(*p);
                p++;
            }
        }
    }
}
```

Переносимость программ на языке C/C++ и на платформе C#/.NET

- Типы данных в C/C++:
 - **char** c1; // символ знаковый или беззнаковый, 1 байт
 - **wchar_t** wc1; // символ беззнаковый, 2 байта (Windows) или 4 байта (Linux)
 - **int** n1; // знаковое целое, 2 байта (16 битов) или 4 байта (32 бита)
 - **long** n2; // знаковое целое, 4 байта (32 бита)
 - **long long** n3; // знаковое целое, 8 байт (64 бита)
- Типы данных в C#/.NET
 - **char** c1; эквивалентно **System.Char** c1; // символ UTF-16, 2 байта
 - **byte** b1; эквивалентно **System.Byte** b1; // знаковое целое, 1 байт (8 битов)
 - **short** n1; эквивалентно **System.Int16** n1; // знаковое целое, 2 байта (16 битов)
 - **int** n2; эквивалентно **System.Int32** n2; // знаковое целое, 4 байта (32 бита)
 - **long** n3; эквивалентно **System.Int64** n3; // знаковое целое, 8 байтов (64 бита)
- Переносимость кода и данных в C/C++:
 - Код программы переносим на уровне исходных текстов
 - Данные на внешних носителях не переносимы между аппаратными платформами
- Переносимость кода и данных в C#/.NET:
 - Код программы переносим на уровне промежуточного кода IL (в компилированном виде)
 - Данные на внешних носителях переносимы между аппаратными платформами

Системные возможности платформы C#/.NET, отсутствующие в языке C/C++

```
class Program
{
    public static int TakePercentChecked(int value, int percent)
    {
        checked // В языке C#, в отличие от C++, имеется контроль переполнения
        {
            return value * percent / 100;
        }
    }

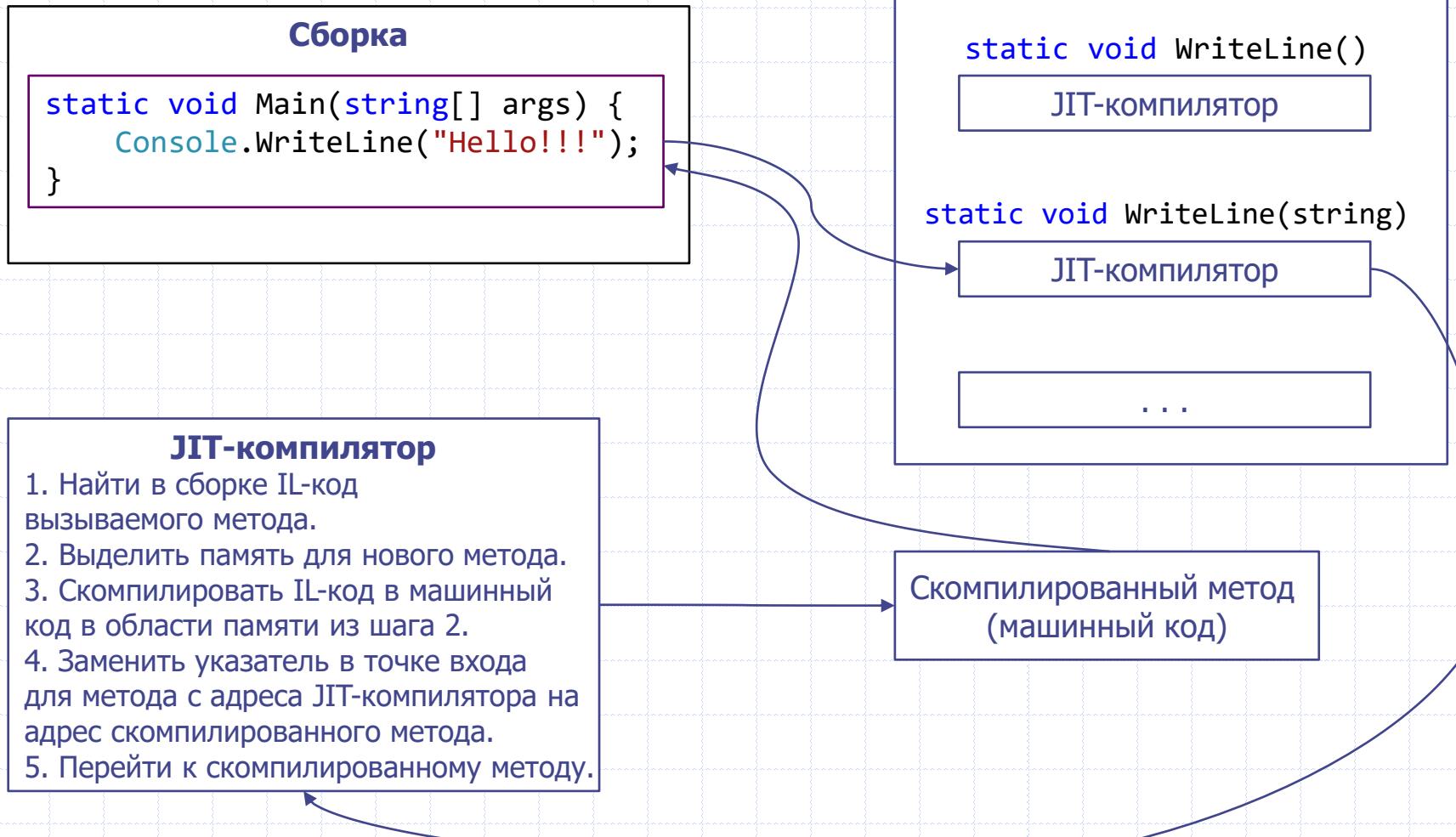
    public static int TakePercentUnchecked(int value, int percent)
    {
        unchecked
        {
            return value * percent / 100;
        }
    }

    static void Main(string[] args)
    {
        try
        {
            int percent = TakePercentUnchecked(int.MaxValue, 50);
            Console.WriteLine(percent); // Результат 0
            percent = TakePercentChecked(int.MaxValue, 50); // Исключение
            Console.WriteLine(percent); // Этот оператор никогда не выполнится
        }
        catch (OverflowException ex)
        {
            Console.WriteLine(ex);
        }
    }
}
```

Динамическая компиляция

- «На лету» (англ. just-in-time – JIT)
 - Выполняется по мере загрузки модулей (сборок) и вызова процедур (методов)
- Заранее или в момент установки программы (англ. ahead-of-time – AOT)
 - Выполняется утилитами NGEN (.NET Framework) или CrossGen (.NET Core, .NET 5)

Динамическая компиляция



Динамическая компиляция

Сборка

```
static void Main(string[] args) {  
    Console.WriteLine("Hello!!!");  
}
```

JIT-компилятор

1. Найти в сборке IL-код вызываемого метода.
2. Выделить память для нового метода.
3. Скомпилировать IL-код в машинный код в области памяти из шага 2.
4. Заменить указатель в точке входа для метода с адреса JIT-компилятора на адрес скомпилированного метода.
5. Перейти к скомпилированному методу.

Класс Console в памяти

```
static void WriteLine()
```

JIT-компилятор

```
static void WriteLine(string)
```

Скомпилированный метод
(машичный код)

...

Модульное программирование – сборки (Assembly)

- Сборка = модуль. Состав модуля
- Пример модуля на языке C#
- Управление версиями, строго именованные сборки
- Глобальный кэш сборок (GAC) — .NET Framework
- Runtime package store — .NET Core, .NET 5+

Объектная модель в среде .NET и языке C#

- Общая система типов данных
- Тип данных Object
- Сылочные и скалярные (value-type) типы данных
- Упаковка и распаковка скалярных типов данных в среде

Тип данных Object

```
public class Object
{
    public Object();
    ~Object(); // virtual void Finalize();

    public static bool Equals(Object objA, Object objB);
    public static bool ReferenceEquals(Object objA, Object objB);
    public virtual bool Equals(Object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
    protected Object MemberwiseClone();
}
```

Ссылочные и скалярные (value-type) типы данных

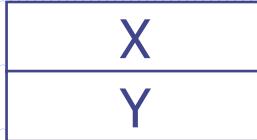
```
public struct Point
{
    public int X;
    public int Y;

    public override string ToString()
    {
        return string.Format("{0},{1}", X, Y);
    }
}
```

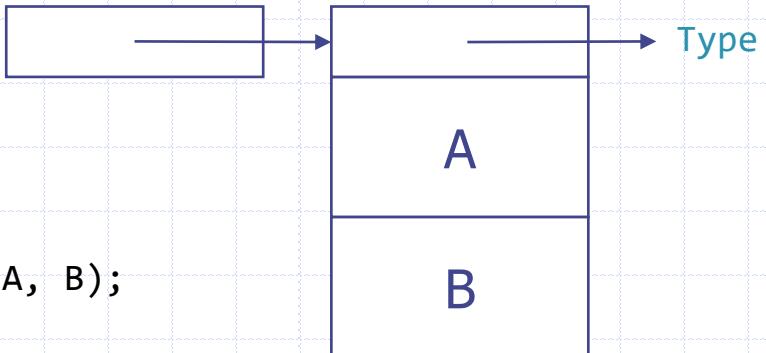
```
public class Rectangle
{
    public Point A;
    public Point B;

    public override string ToString()
    {
        return string.Format("[{0};{1}]", A, B);
    }
}
```

```
Point p = new Point();
```

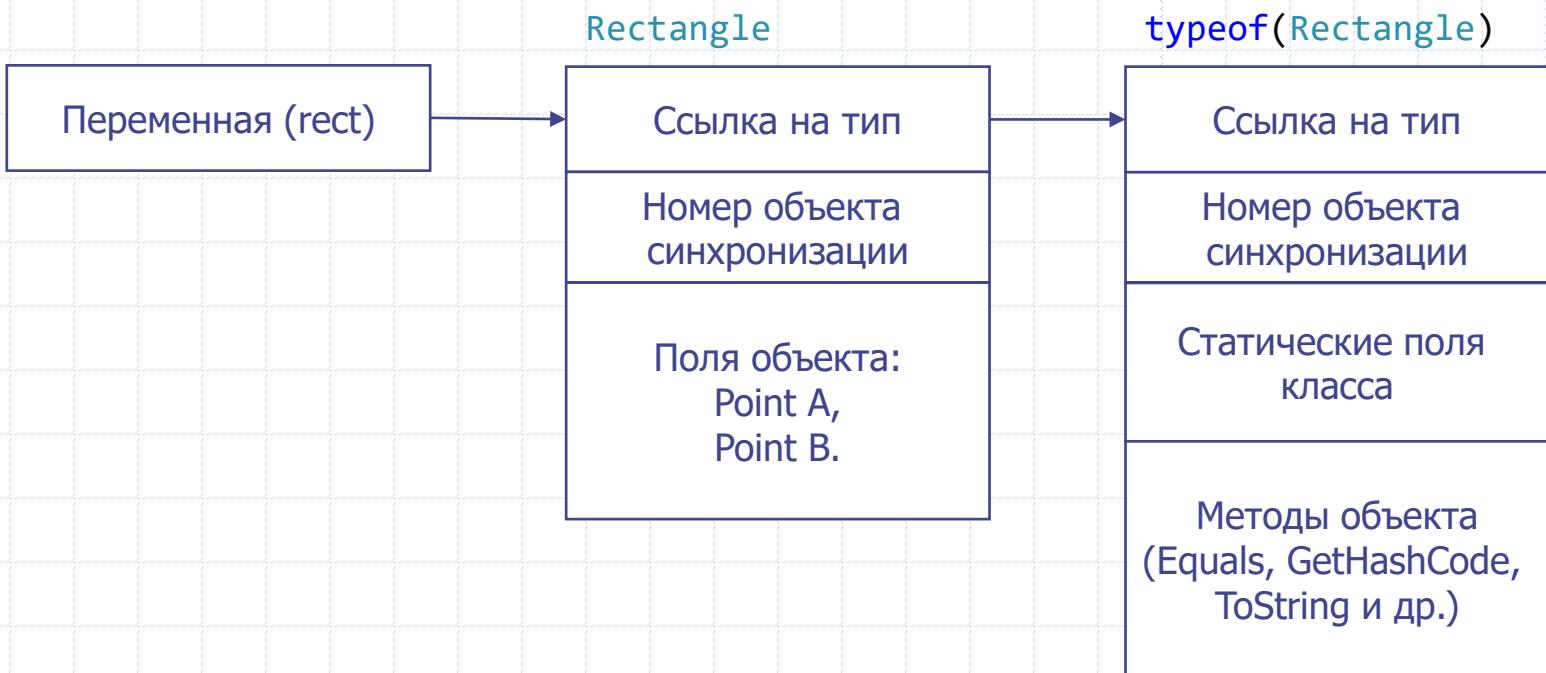


```
Rectangle r = new Rectangle();
```



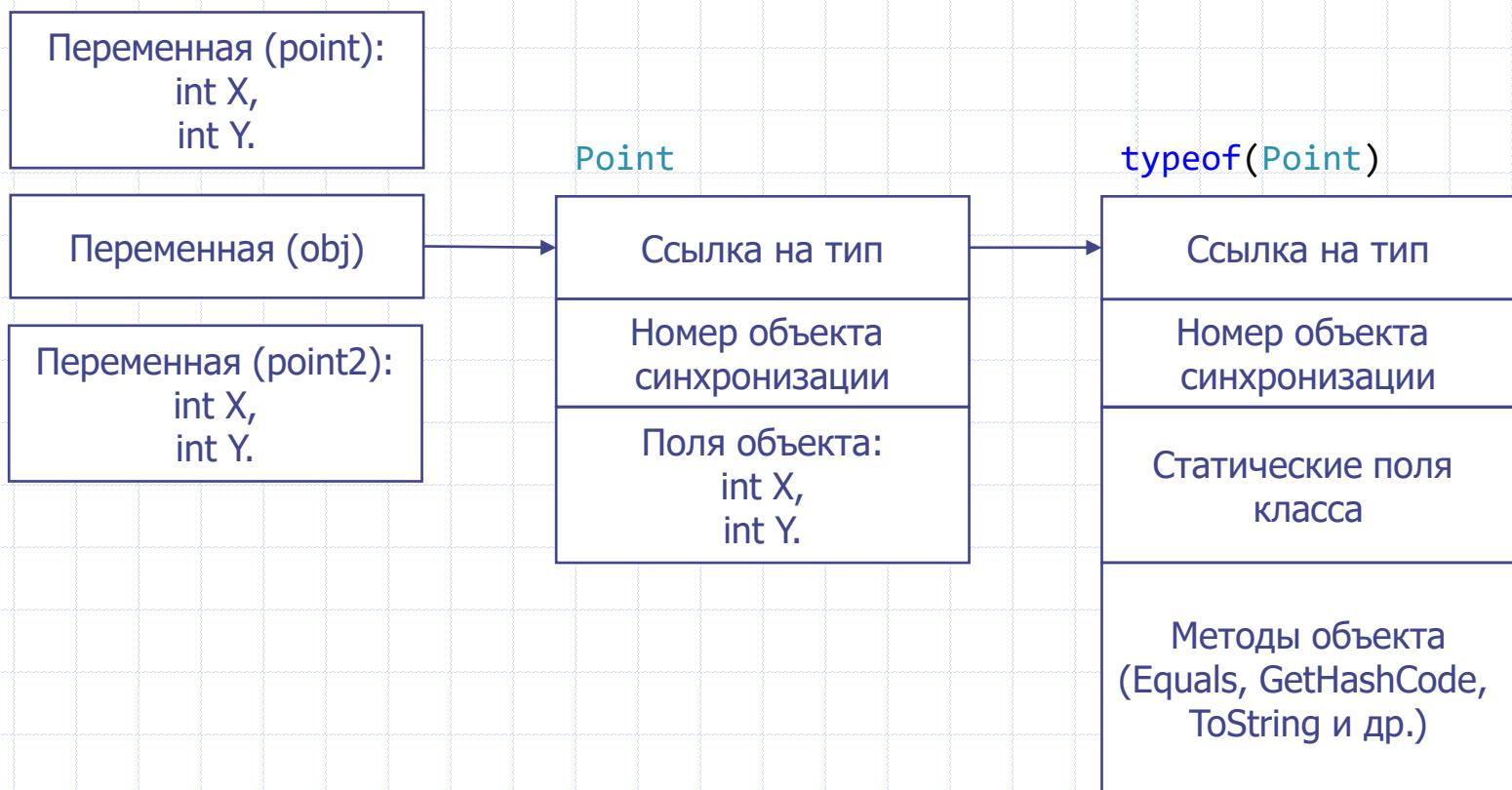
Представление ссылочных типов данных в динамической памяти

```
Rectangle rect = new Rectangle();
```



Представление скалярных типов данных в памяти. Упаковка и распаковка

```
Point point = new Point();  
object obj = point;  
Point point2 = (Point)obj;
```



Упаковка и распаковка скалярных типов данных

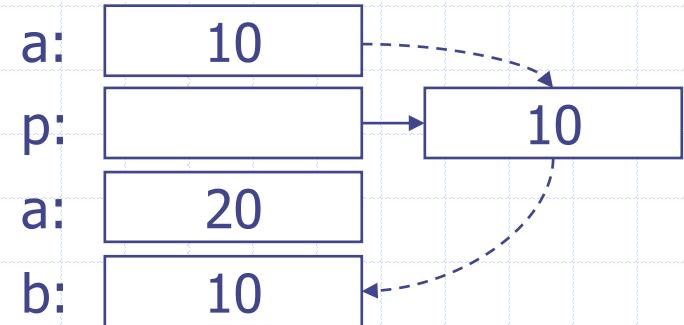
```
static void Main(string[] args)
```

```
{  
    int a = 10;  
    object p = a;  
    a = 20;  
    int b = (int)p;
```

```
Console.WriteLine("a = {0}", a);
```

```
Console.WriteLine("b = {0}", b);
```

```
}
```



ВЫЗОВ ПЕРЕОПРЕДЕЛЕННОГО ВИРТУАЛЬНОГО МЕТОДА ДЛЯ СКАЛЯРНЫХ ТИПОВ ДАННЫХ

```
static string GetString(Point p)    public struct Point
{
    return p.ToString();           {
}
}

.method private hidebysig static
string GetString(
    valuetype Example1.Point p
) cil managed
{
    .maxstack 8

    IL_0000: ldarga.s p
    IL_0002: constrained. Example1.Point
    IL_0008: callvirt instance string [System.Runtime]System.Object::ToString()
    IL_000d: ret
}
```

```
public struct Point
{
    public int X;
    public int Y;

    public override string ToString()
{
    return string.Format(
        "({0},{1})", X, Y);
}
```

Доступ к метаданным во время работы программы

```
public class Object
{
    static void Main(string[] args)
    {
        object r = new Rectangle();

        Type t = r.GetType();
        Console.WriteLine(t.Name);

        MemberInfo[] members = t.GetMembers();
        foreach (MemberInfo m in members)
        {
            Console.WriteLine(m.ToString());
        }
    }
}
```

Пользовательские метаданные на примере автоматизации тестирования

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = false)]
public class TestMethodAttribute : Attribute
{
    public TestMethodAttribute()
    {}

    public TestMethodAttribute(string category)
    {
        Category = category;
    }

    public string Category { get; set; }
    public int Priority { get; set; }
}

public class Tests
{
    [TestMethod("Функциональность", Priority = 1)]
    // var attr = new TestMethodAttribute("Функциональность");
    // attr.Priority = 1;
    static void TestOne()
    {
        Console.WriteLine(nameof(TestOne));
    }

    [TestMethod("Функциональность", Priority = 1)]
    static void TestTwo()
    {
        Console.WriteLine(nameof(TestTwo));
    }
}
```

Пользовательские метаданные на примере автоматизации тестирования (продолжение)

```
public class Program
{
    static List<MethodInfo> GetTestMethods(Assembly assembly)
    {
        List<MethodInfo> testMethods = new List<MethodInfo>();
        Type[] types = assembly.GetExportedTypes();
        foreach (Type type in types)
        {
            MethodInfo[] methods = type.GetMethods(
                BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Static);
            foreach (MethodInfo method in methods)
            {
                TestMethodAttribute[] attributes = (TestMethodAttribute[])
                    .GetCustomAttributes(typeof(TestMethodAttribute), false);
                if (attributes != null &amp; attributes.Length > 0)
                    testMethods.Add(method);
            }
        }
        return testMethods;
    }

    static void Main(string[] args)
    {
        Assembly assembly = Assembly.GetExecutingAssembly();
        List<MethodInfo> methods = GetTestMethods(assembly);
        foreach (MethodInfo method in methods)
        {
            Action testAction = (Action)Delegate.CreateDelegate(typeof(Action), method);
            testAction();
        }
    }
}
```

Модели утилизации динамической памяти

- Модель с явным освобождением памяти
- Модель со счетчиками ссылок
- Модель с иерархией владения
- Модель с владеющими ссылками
- Модель с автоматической сборкой мусора
- Модель с автоматической сборкой мусора и явным освобождением памяти

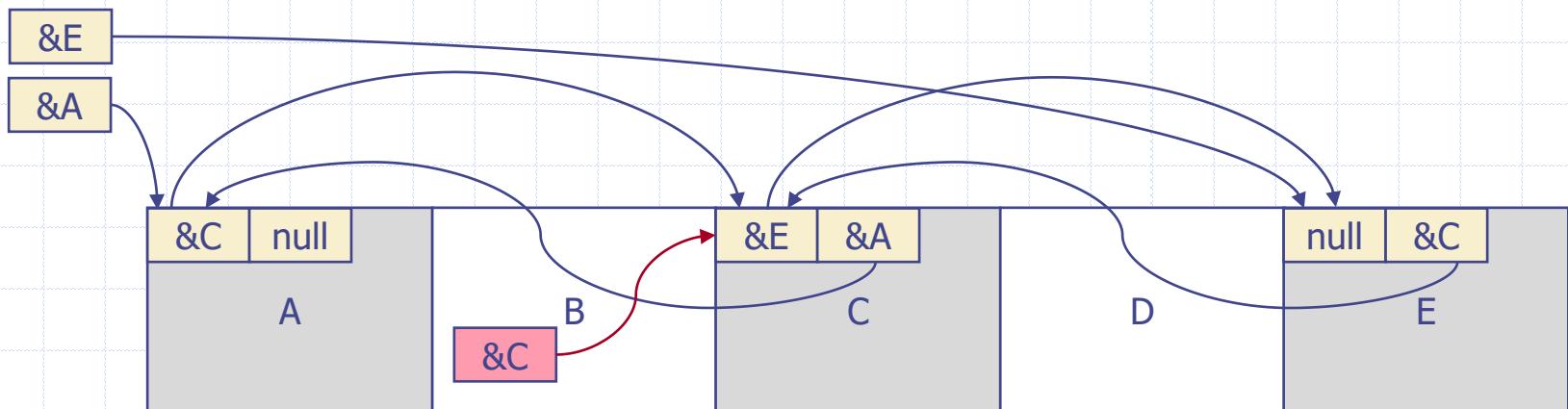
Модель с явным освобождением памяти

- Программист сам заботится о выделении и утилизации объектов
- malloc и free в C, new и delete в C++
- Список свободных блоков обычно двусвязный и хранится внутри свободной памяти
- Достоинство – детерминизм.
 - Предсказуемы временные задержки на выделение и освобождение памяти
 - Предсказуем порядок работы конструкторов и деструкторов и связанные с этим накладные расходы
- Недостаток – ненадежность и подверженность ошибкам:
 - «Утечка» памяти
 - «Зависание» ссылок
 - Повторное удаление объектов
 - Сильная фрагментация (объекты нельзя перемещать в адресном пространстве)

Модель с явным освобождением памяти.

Проблемы применения

- Менеджер памяти использует память уничтоженных объектов для организации списка свободных блоков.
- «Зависшая» ссылка – ссылка на уничтоженный объект. Обращение по «зависшей» ссылке как правило портит список свободных блоков менеджера памяти.
 - Возникающие сбои происходят не сразу, а спустя некоторое время, когда уже непонятно, какая подпрограмма нарушила целостность данных менеджера памяти.
- Повторное удаление объекта тоже портит список свободных блоков.



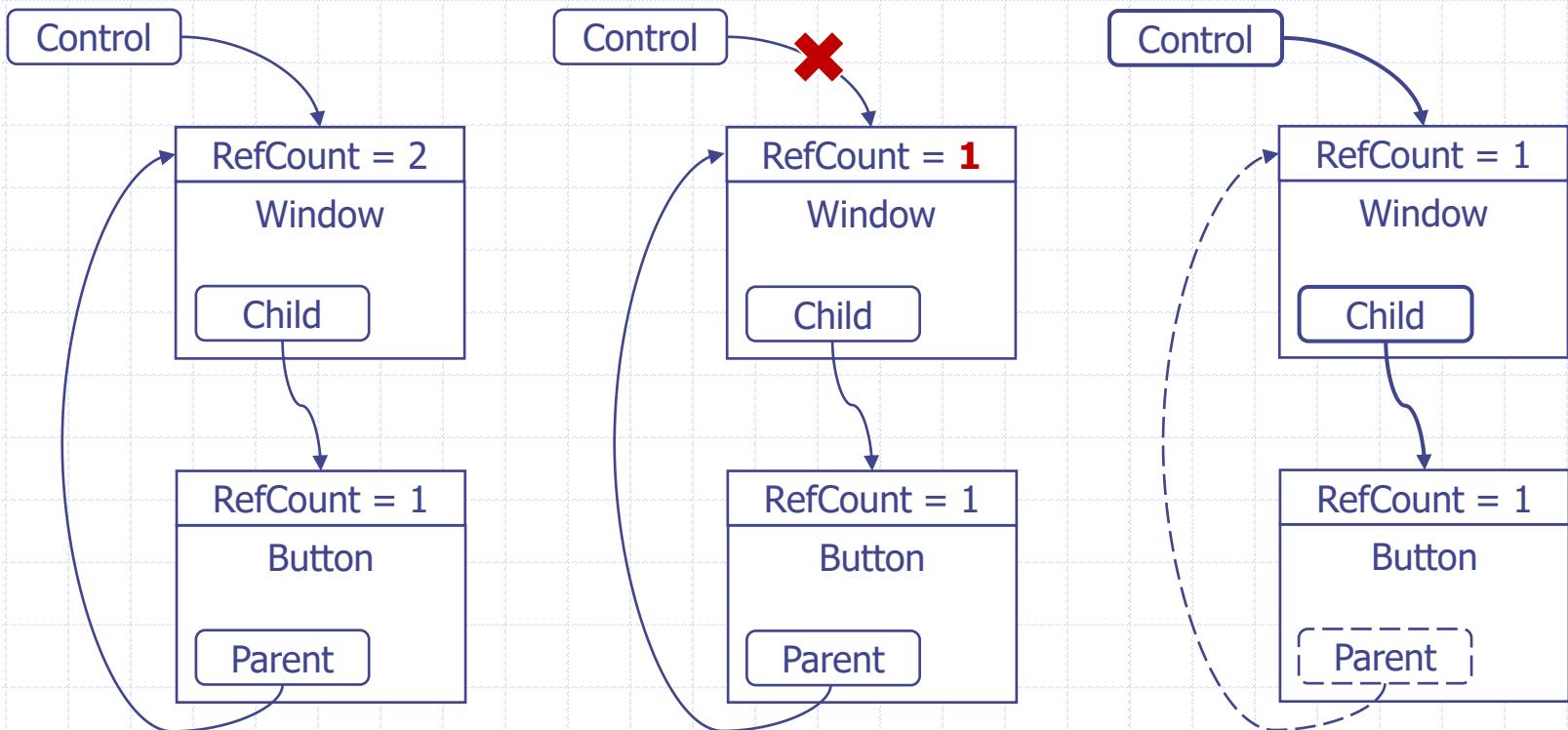
Модель со счетчиками ссылок

- Обычно надстраивается над моделью с явным освобождением памяти
- С каждым объектом ассоциируется целочисленный счетчик ссылок
 - Счётчик ссылок хранится либо в одном из полей объекта, либо «навешен» снаружи.
 - При создании объекта счетчик устанавливается в значение 0.
 - Счётчик увеличивается на единицу при создании каждой новой ссылки на объект: присваивание переменной и передача ссылки в аргументах подпрограммы.
 - При пропадании каждой ссылки значение счетчика уменьшается на единицу.
 - Когда значение счётчика становится равным нулю, объект уничтожается (оператором `delete`).
 - Применяется в технологии СОМ и в ряде языков программирования (Object Pascal, Objective-C, Swift, Python).
- Для решения проблемы циклических связей ссылки делят на «сильные» и «слабые».
 - Сильные ссылки влияют на счетчик ссылок, а слабые ссылки — нет.
 - При уничтожении объекта слабые ссылки автоматически обнуляются.
 - Для доступа к объекту слабую ссылку нужно предварительно превратить в сильную ссылку.
 - Применяется в языках Objective-C, Swift.
- Для решения проблемы циклических связей также может использоваться вспомогательный сборщик мусора.
 - Применяется в языке Python.

Модель со счетчиками ссылок.

Проблема применения

- Достоинство – детерминизм.
- Недостатки
 - Дополнительные накладные расходы на элементарное копирование ссылок.
 - Ошибки в выборе между «сильными» и «слабыми» ссылками приводят либо к утечкам памяти, либо к преждевременному уничтожению объектов.

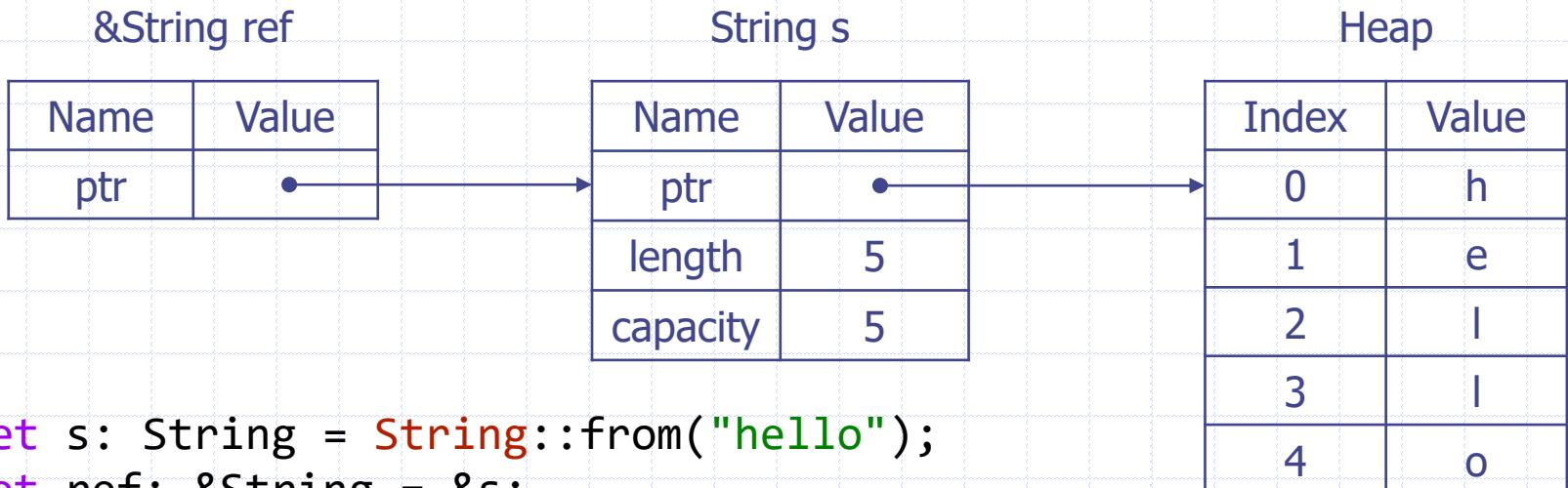


Модель с иерархией владения

- Обычно надстраивается над моделью с явным освобождением памяти
- Каждому объекту при создании назначается объект-владелец
 - Владелец отвечает за уничтожение подчиненных объектов.
 - Можно не заботиться о том, что ссылки на объект пропадут, и произойдет утечка памяти.
 - Объект можно уничтожить принудительно, даже если у него есть владелец. При этом объект либо изымается из списка подчиненных объектов своего владельца, либо помечается как уничтоженный для предотвращения повторного уничтожения.
 - Объект может быть создан без владельца, в этом случае он требует явного уничтожения. Это ничем не отличается от уже рассмотренной модели с ручным освобождением памяти.
- Применяется:
 - В библиотеках визуального программирования (для управления компонентами)
 - Может совмещаться с моделью на основе счетчиков ссылок. Такая гибридная модель используется, например, в новейшей технологии драйверов для ОС Windows.
- Достоинства
 - Значительно сокращает риск утечек памяти.
 - Позволяет более успешно бороться с зависшими указателями, например, путем рассылки сообщений об уничтожении объектов по иерархии. Обрабатывая эти сообщения, объекты-получатели могут обнулять сохраненные ссылки на уничтожаемые объекты.
- Недостатки:
 - Программист полностью не избавляется от необходимости явно освобождать память
 - Не решается проблема фрагментации памяти

Модель с владеющими ссылками

- Ссылки «владеют» своими объектами: у каждого объекта в один момент времени может быть только один владелец.
- Присваивание ссылки имеет семантику передачи владения.
- Компилятор следит за существованием владельца объекта и за передачей владения между ссылками.
 - Когда владеющая ссылка пропадает (например, выходит из области видимости), объект уничтожается.
 - Попытка использовать предыдущую ссылку после передачи владения приводит к ошибке компиляции.
- Можно создавать «невладеющие» ссылки, которые в реализации являются ссылками на владеющие.
- Применяется в языке программирования Rust. Сильно усложняет программирование.



Модель с автоматической сборкой мусора

- Предусматривает лишь возможность создавать объекты, но не уничтожать их.
- Система сама следит за тем, на какие объекты еще имеются ссылки, а на какие — уже нет.
- Когда объекты становятся недостижимы через имеющиеся в программе ссылки (превращаются в «мусор»), их память автоматически возвращается системе.
- Сборка мусора происходит в две фазы:
 - Сначала сборщик мусора находит все достижимые по ссылкам объекты и помечает их. Обход графа достижимых объектов начинается от «корней», к которым относятся все глобальные ссылки и ссылки в стеках имеющихся программных потоков. Анализируя метаданные (информацию о типах данных, которая размещается внутри выполняемых модулей), сборщик мусора выясняет, где внутри объектов имеются ссылки на другие объекты. Следуя по этим ссылкам, сборщик мусора обходит все цепочки объектов и выясняет, какие блоки памяти стали свободными.
 - Затем сборщик мусора перемещает объекты в адресном пространстве программы (с соответствующей корректировкой значений ссылок) для устранения фрагментации памяти.
- Достоинства
 - Нет утечек памяти, нет фрагментации памяти, нет зависших указателей.
 - По скорости выделения памяти данная модель сравнима со стеком, ведь выделение объекта — это, по сути, увеличение указателя свободной области памяти на размер размещаемого объекта.
- Недостатки
 - Периодическое устранение фрагментации может требовать много времени и привести к ощутимой задержке в работе программы. Плохо подходит для задач реального времени.
 - Легальная утечка ресурсов из-за забытых регистраций обработчиков событий. Программисты порой забывают отключать обработчики событий, в результате ассоциированные объекты остаются в памяти, несмотря на кажущееся отсутствие ссылок на них в программе.

Амортизация задержек в сборщиках мусора

- В среде .NET
 - Вся память делится на поколения (например, младшее, среднее, старшее). Объекты создаются в младшем поколении и перемещаются в среднее поколение, а затем в старшее поколение, пережив сборку мусора. Сборка мусора выполняется не во всей памяти, а лишь в тех поколениях, в которых исчерпалось свободное место.
- В среде Java (JRTS — Java Real-Time Specification)
 - Дефрагментация выполняется эпизодически и лишь в самом крайнем случае, когда не может быть найден свободный блок памяти нужного размера. Кроме того, сборка мусора выполняется в течение фиксированных интервалов времени (квантов), которые обязательно чередуются с квантами работы программы.

Механизм сборки мусора.

Поколения объектов

- Принцип поколений основан на наблюдении, что объекты, создаваемые раньше, как правило, живут дольше.
- Количество поколений соответствует числу уровней кэширования с учетом ОЗУ. В современных архитектурах обычно три поколения:
 - Поколение 0 соответствует кэшу процессора, ~ 256 КБ;
 - Поколение 1 соответствует кэшу ОЗУ, ~ 2 МБ;
 - Поколение 2 соответствует ОЗУ, ~ 10 МБ.
- Объекты создаются в младшем поколении 0 и перемещаются в среднее поколение 1, а позже в старшее поколение 2, пережив сборку мусора.
- Сборка мусора выполняется не во всей памяти, а лишь в тех поколениях, в которых исчерпалось свободное место — чаще в нулевом, реже в первом, и еще реже во втором поколении.
- Сборка мусора выполняется часто, задержек при сборке мусора много, но их средняя длительность небольшая.

Механизм сборки мусора. Поколения объектов (продолжение)

- Все новые объекты создаются в поколении 0 (изначально пустое).
- Когда при очередном создании объекта оказывается, что поколение 0 исчерпало свой запас, запускается сборка мусора.

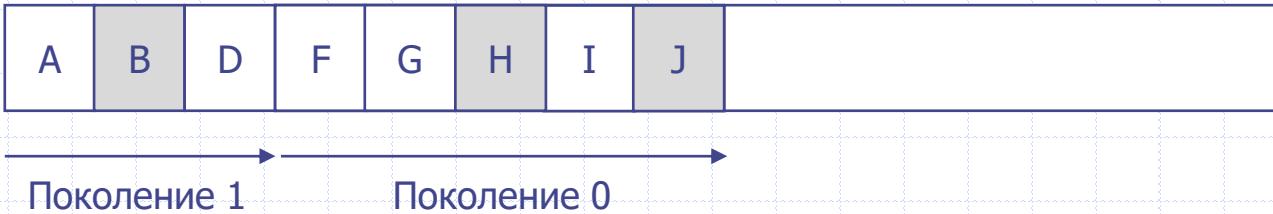


- Достижимые объекты уплотняются и попадают в поколение 1.
- Поколение 0 становится пустым и смещается в адресном пространстве в сторону больших адресов.



Механизм сборки мусора. Поколения объектов (продолжение)

- Новые объекты создаются в поколении 0, и когда оно опять оказывается исчерпано, снова запускается сборка мусора.

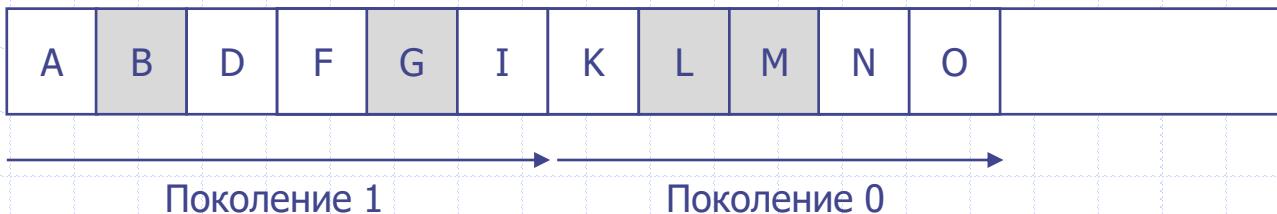


- Сборка мусора выполняется только в исчерпавшихся поколениях.
- Сборка мусора в поколении 1 выполняется только при сборке мусора в поколении 0 при условии, что при уплотнении и логическом перемещении объектов в поколение 1 это поколение исчерпало свой запас.

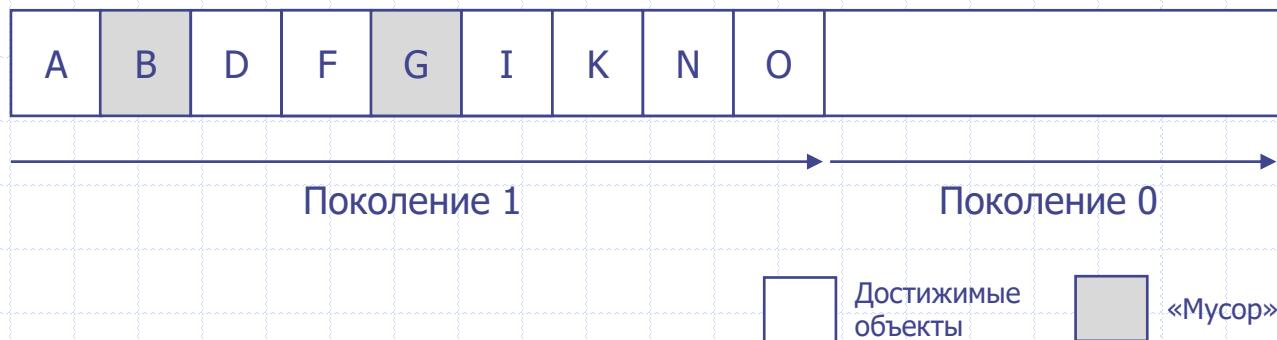


Механизм сборки мусора. Поколения объектов (продолжение)

- Новые объекты создаются в поколении 0, и когда оно опять оказывается исчерпано, снова запускается сборка мусора.

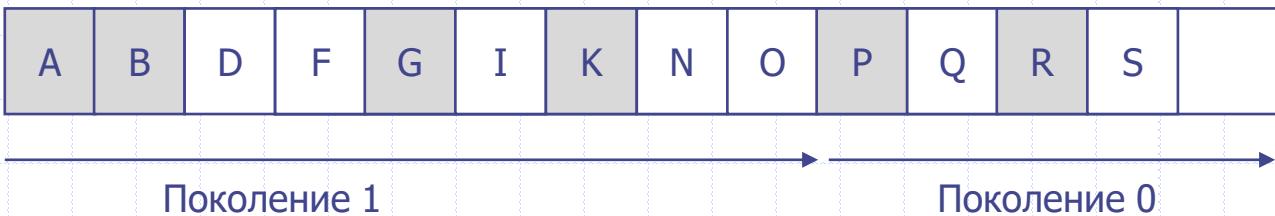


- Сборка мусора выполняется только в исчерпавшихся поколениях.
- Сборка мусора в поколении 1 выполняется только при сборке мусора в поколении 0 при условии, что при уплотнении и логическом перемещении объектов в поколение 1 это поколение исчерпало свой запас.



Механизм сборки мусора. Поколения объектов (продолжение)

- Новые объекты создаются в поколении 0, и когда оно опять оказывается исчерпано, снова запускается сборка мусора.



- Сборка мусора выполняется только в исчерпавшихся поколениях.
- Сборка мусора в поколении 2 выполняется только при сборке мусора в поколении 1 при условии, что при уплотнении и логическом перемещении объектов в поколение 2 это поколение исчерпало свой запас.



Механизм сборки мусора.

Принципы работы

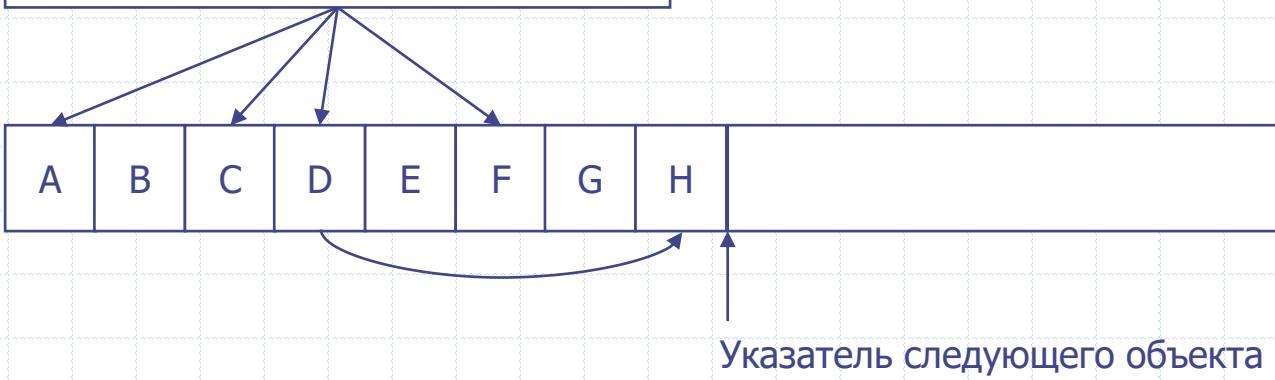
- Размеры поколений могут меняться сборщиком мусора по ходу работы программы на основе анализа количества переживающих сборку мусора объектов.
- В поколение 0 попадают только «небольшие» объекты, размер которых меньше 85000 байтов.
 - «Большие» объекты сразу попадают в поколение 2
- «Серверный» сборщик мусора создаёт отдельное поколение 0 на каждый программных поток.
 - Позволяет избежать синхронизации программных потоков при выделении памяти оператором new.
- Возможна параллельная фоновая сборка мусора без остановки всех программных потоков.

Механизм сборки мусора и завершения объектов (метод Finalize)

- Объекты, управляющие какими-то ресурсами, могут иметь метод Finalize (так называемый деструктор в языке C#) для завершения времени жизни объектов и освобождения ресурсов.
 - Метод Finalize или, говоря иначе, деструктор выполняется асинхронно в контексте сборщика мусора.
 - Не определён момент вызова метода Finalize. Вызов может быть отложен вплоть до завершения программы.
 - Не определён порядок вызова методов Finalize разных объектов. Метод Finalize вложенного объекта может вызваться раньше метода Finalize объекта-хозяина.
 - Нет гарантии, что метод Finalize вообще будет вызван. Если происходит завершение программы, то каждому методу Finalize даётся лишь 2 секунды, а всем методам вместе – лишь 40 секунд, на выполнение.

Механизм сборки мусора и завершения объектов (метод Finalize)

Корни:
Статические поля
Параметры методов (на стеке)
Локальные переменные (на стеке)
Регистры процессора

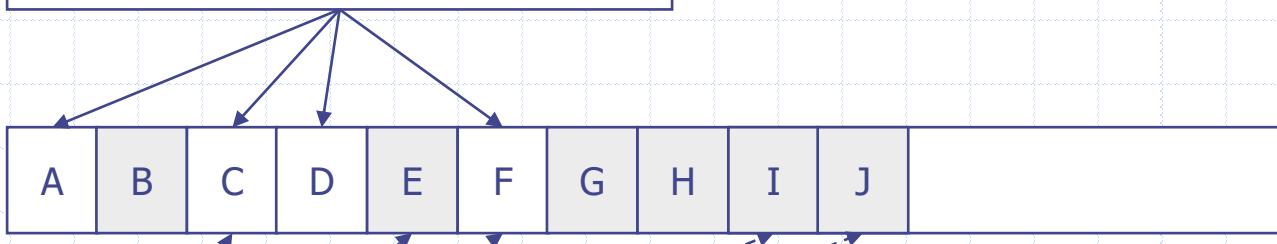


Механизм сборки мусора и завершения объектов (метод Finalize)

- В момент создания оператором new объекты с реализованным методом Finalize помещаются в список завершаемых объектов (Finalization list).
- Когда такие объекты становятся недостижимы по имеющимся указателям в программе, сборщик мусора утилизирует их особым образом. Объект перемещается в старшее поколение (например, из нулевого в первое), а ссылка на объект из списка Finalization list переносится в очередь достижимых объектов (Freachable queue), делая объект временно достижимым.
- Очередь Freachable queue обрабатывается выделенным высокоприоритетным потоком, который удаляет объект из очереди и вызывает у него метод Finalize. Ссылка на объект в псевдо-параметре this во время работы метода является одним из корней, удерживающих объект от превращения в мусор.
- После завершения метода Finalize объект окончательно превращается в мусор и дальше утилизируется так же, как и обычные объекты.
- Метод Finalize не может быть вызван в контексте сборщика мусора (минута очередь Freachable и высокоприоритетный поток, который её обслуживает), поскольку метод Finalize это обычный метод, который может создавать новые объекты оператором new, а значит вызов оператора new приводил бы к рекурсивному вызову сборки мусора в условиях отсутствия места в нулевом поколении.

Механизм сборки мусора и завершения объектов (метод Finalize)

Корни:
Статические поля
Параметры методов (на стеке)
Локальные переменные (на стеке)
Регистры процессора



Список завершаемых объектов
(Finalization list)



Очередь достижимых объектов
(Reachable queue)



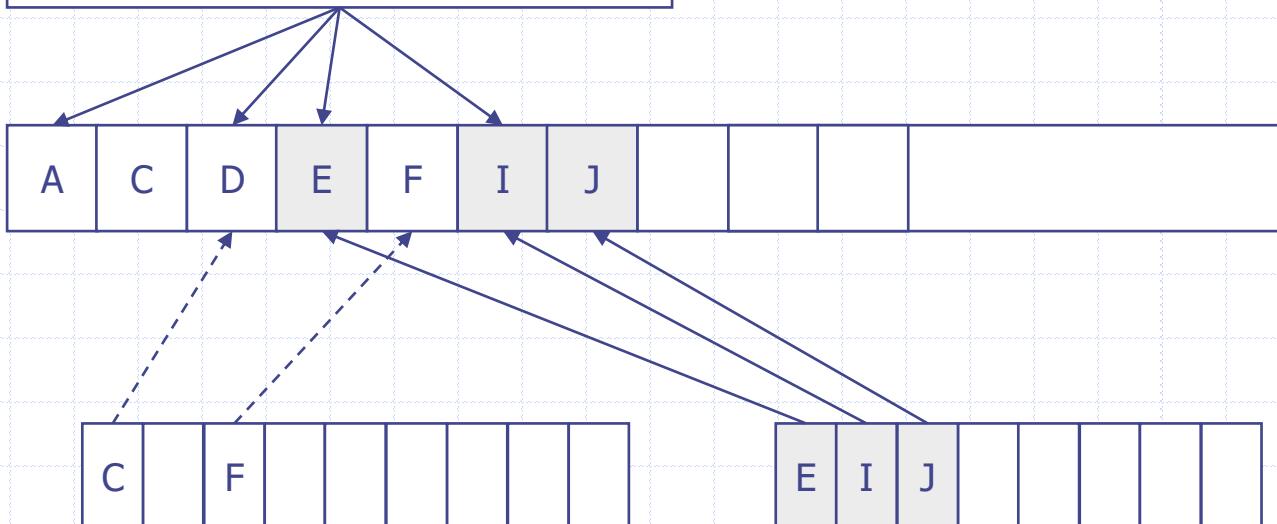
Достижимые
объекты



«Мусор»

Механизм сборки мусора и завершения объектов (метод Finalize)

Корни:
Статические поля
Параметры методов (на стеке)
Локальные переменные (на стеке)
Регистры процессора



Список завершаемых объектов
(Finalization list)

Очередь достижимых объектов
(Reachable queue)

Доступные
объекты

«Мусор»

Проблема метода Finalize и её решение – интерфейс IDisposable

- Метод Finalize порождает проблему, если объект управляет некоторым ресурсом, например, сетевым соединением. Открытие соединения происходит при создании и инициализации объекта, т.е. предсказуемо, а закрытие соединения — во время сборки мусора, т.е. непредсказуемо и далеко не сразу после потери последней ссылки на объект. В результате лимит сетевых соединений, или других ресурсов, может временно исчерпаться.
- Для решения указанной проблемы в среде .NET используется детерминированное завершение жизни объектов через интерфейс IDisposable.

Интерфейс IDisposable

- Для детерминированное завершение жизни объектов в среде .NET используется интерфейс IDisposable.
- Этот интерфейс имеет единственный метод Dispose, который реализуется в объектах, управляющих ресурсами.
- Метод Dispose как правило освобождает ресурсы и отменяет работу процедуры-завершителя (метода Finalize), чтобы ускорить освобождение памяти.
- После вызова метода Dispose объект не уничтожается, а остается в памяти до тех пор, пока не пропадут все ссылки на него.

Пример с методом Finalize

```
public class LogFile : Object
{
    private StreamWriter writer;

    public LogFile(string filePath)
    {
        writer = new StreamWriter(filePath, append: true);
    }

    ~LogFile() // protected override void Finalize()
    {
        Console.WriteLine("Выполняется LogFile.Finalize");
        // Thread.Sleep(3000);
        writer.Close();
        Console.WriteLine("LogFile.Finalize выполнился");
    }

    public void Write(string str)
    {
        writer.Write(str);
    }
}
```

Пример с интерфейсом IDisposable

```
public class LogFile : Object, IDisposable
{
    private StreamWriter writer;

    public LogFile(string filePath)
    {
        writer = new StreamWriter(filePath, append: true);
    }

    public void Dispose()
    {
        writer.Close();
    }

    public void Write(string str)
    {
        writer.Write(str);
    }
}
```

Проблемы интерфейса IDisposable

- Проблемы интерфейса IDisposable:
 - После вызова метода Dispose в программе могут оставаться ссылки на объект, находящийся уже в некорректном состоянии. Программе никто не запрещает обращаться по этим физически доступным, но логически зависимым ссылкам и вызывать у некорректного объекта различные методы.
 - Метод Dispose может вызываться повторно, в том числе рекурсивно.
 - В программе с несколькими вычислительными потоками может происходить асинхронный вызов метода Dispose для одного и того же объекта.
- Для решения проблем метода Dispose программистам было предписано делать следующее:
 - 1) определять в объекте булевский флаг, позволяющий выяснить, работал ли в объекте код завершения, и игнорировать повторные вызовы метода Dispose, проверяя упомянутый булевский флаг;
 - 2) в программах с несколькими вычислительными потоками блокировать объект внутри метода Dispose на время работы кода завершения;
 - 4) в начале public-методов проверять, что объект уже находится в завершенном состоянии, и в этом случае создавать исключение класса ObjectDisposedException.

Пример с интерфейсом IDisposable (правильный)

```
public class LogFile : Object, IDisposable
{
    private StreamWriter writer;
    private bool disposed;

    public LogFile(string filePath)
    {
        writer = new StreamWriter(filePath, append: true);
    }

    public void Dispose()
    {
        if (!disposed)
        {
            writer.Close();
            disposed = true;
        }
    }

    public void Write(string str)
    {
        if (disposed)
            throw new ObjectDisposedException(this.ToString());

        writer.Write(str);
    }
}
```

Использование объектов, реализующих интерфейс IDisposable

```
LogFile f = new LogFile("Log.txt");
try
{
    f.Write("Ключ на старт");
    f.Write("Протяжка-1");
    f.Write("Продувка");
}
finally
{
    f.Dispose();
}

// Короткий эквивалент в C# с оператором using:
```

```
using (var f = new LogFile("Log.txt"))
{
    f.Write("Ключ на старт");
    f.Write("Протяжка-1");
    f.Write("Продувка");
}
```

Пример с методом Finalize и интерфейсом IDisposable

```
public class NativeBuffer : Object,
    IDisposable
{
    private IntPtr handle;
    private bool disposed;

    public NativeBuffer(int size)
    {
        handle = Marshal.AllocHGlobal(size);
    }

    ~NativeBuffer()
    {
        Dispose(false);
    }

    public IntPtr Handle
    {
        get
        {
            if (!disposed)
                return handle;
            else
                throw new ObjectDisposedException(ToString());
        }
    }

    public void Dispose()
    {
        if (!disposed)
        {
            Dispose(true);
            GC.SuppressFinalize(this);
            disposed = true;
        }
    }

    protected virtual
    void Dispose(bool disposing)
    {
        if (handle != IntPtr.Zero)
            Marshal.FreeHGlobal(handle);
    }
}
```

Пример с методом Finalize и интерфейсом IDisposable

- Если класс одновременно реализует метод Finalize и интерфейс IDisposable, то в методе Dispose нужно отменять вызов метода Finalize путём вызова метода GC.SuppressFinalize.
- Если вызов GC.SuppressFinalize пропущен, то ошибки в работе программы не будет, но эффективность работы с памятью ощутимо снизится, поскольку сборщик мусора вынужден будет переносить уже разрушенный объект в старшее поколение для бесполезного вызова метода Finalize.

Принудительная сборка мусора

```
GC.Collect(  
    generation: GC.MaxGeneration,  
    mode: GCCollectionMode.Forced,  
    blocking: true,  
    compacting: true);  
  
GC.WaitForPendingFinalizers();  
  
GC.Collect();  
  
// Сборка мусора в больших объектов (Large Object Heap - LOH):  
GCSettings.LargeObjectHeapCompactionMode =  
    GCLargeObjectHeapCompactionMode.CompactOnce;  
GC.Collect();
```

«Слабые» ссылки (WeakReference)

```
class Program
{
    static void Example()
    {
        Object o = File.ReadAllLines(@"C:\MyTestLog.txt");

        // Работа с о

        WeakReference wr = new WeakReference(o);
        o = null;

        // Работа с другими данными

        // Сборка мусора
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        Object o2 = wr.Target;
        if (o2 == null)
        {
            // Памяти не хватило, объект был утилизирован
            o2 = File.ReadAllLines(@"C:\MyTestLog.txt");
            wr.Target = o2;
        }

        // Работа с o2

        o2 = null;
    }
}
```

Модель с автоматической сборкой мусора и явным освобождением памяти

- Перспективная модель, сочетающая два достоинства:
 - быструю автоматическую сборку мусора. Означает, что программист может полагаться на то, что система следит за потерей ссылок на объекты и устраняет утечку памяти;
 - безопасное принудительное освобождение памяти. Означает, что программист вправе уничтожить объект, при этом память объекта возвращается системе, а все имеющиеся на него ссылки становятся недействительными (например, обнуляются).
- На самом деле не нова и уже давно применяется:
 - в компьютерах Эльбрус на основе одноименного процессора;
 - в компьютерах IBM AS/400 на основе процессора PowerPC
- Имеет очень эффективную реализацию за счет аппаратной поддержки.
 - На каждое машинное слово в этих компьютерах отводится два дополнительных бита, называемых битами тегов. Значения этих битов показывают, свободно ли машинное слово, или занято, и если занято, то хранится ли в нем указатель, или скалярное значение. Этими битами управляют аппаратура и операционная система, прикладным программам они недоступны.
 - Программа не может создать ссылку сама, например, превратив в нее число или другие скалярные данные. Созданием объектов занимается система, которая размещает в памяти объекты и создает ссылки на них. При уничтожении объектов соответствующие теги памяти устанавливаются в состояние, запрещающее доступ. Попытка обратиться к свободной памяти по зависшему указателю приводит к аппаратному прерыванию (подобно обращению по нулевому указателю).
 - Поскольку вся память помечена тегами, сборщику мусора нет необходимости анализировать информацию о типах, чтобы разобраться, где внутри объектов располагаются ссылки на другие объекты.
 - Что более важно, сборщику мусора почти не нужно тратить время на поиск недостижимых объектов, поскольку освобожденная память помечена с помощью тех же тегов.

Модель с автоматической сборкой мусора и явным освобождением памяти. Спецификация для языков программирования.

- Выделение динамической памяти выполняется оператором/процедурой `new` (это действие считается элементарным в системе). Выделенная память автоматически инициализируется нулями и всегда привязывается к типу созданного в памяти объекта.
- Уничтожение объекта — освобождение занимаемой им динамической памяти — выполняется автоматически при пропадании всех ссылок на объект. Для дефрагментации освободившихся участков памяти периодически выполняется сборка мусора, в результате которой объекты сдвигаются, а ссылки на них корректируются.
- Объекты можно уничтожать принудительно с помощью оператора/процедуры `delete`. В результате этого действия все ссылки на объект становятся недействительными, а попытка последующего доступа к объекту приводит к исключительной ситуации. Дефрагментация освобожденной этим способом памяти выполняется во время сборки мусора. При этом оставшиеся ссылки корректируются и получают некоторое зарезервированное недействительное значение, например, `-1` (зависшие ссылки можно было бы обнулять, но в этом случае стерлась бы разница между нулевой и зависшей ссылкой, что ухудшило бы диагностику ошибок).

Модель с автоматической сборкой мусора и явным освобождением памяти. Возможные программные реализации.

- Первое простейшее решение состоит в том, чтобы по каждому вызову оператора `delete` выполнять просмотр памяти с корректировкой недействительных ссылок.
 - Просмотр занимает значительно меньше времени, чем полная сборка мусора с дефрагментацией памяти. Решение подходит для мобильных и встроенных устройств с небольшим объемом ОЗУ и без поддержки виртуальной памяти.
- Второе решение основано на использовании средств аппаратной поддержки виртуальной памяти, которая существует в большинстве современных компьютерных архитектур.
 - Виртуальная память практически всегда имеет страничную организацию. Страницы памяти могут быть выгружены на диск и помечены как отсутствующие. Обращение к данным в выгруженной странице приводит к аппаратному прерыванию. Это прерывание обрабатывает ОС, которая подгружает запрошенную страницу с диска и замещает ею одну из редко используемых страниц. В этом механизме нас интересует возможность аппаратно перехватывать обращения к страницам виртуальной памяти. На самом деле страницы могут оставаться в памяти и на диск не выгружаться.
 - Идея состоит в том, чтобы при вызове оператора `delete` помечать страницы виртуальной памяти, в которых располагается удаляемый объект, как отсутствующие. Обращение к данным на этих страницах будет вызывать аппаратное прерывание. Обрабатывая это прерывание, система проверяет, куда именно выполняется обращение: к освобожденному участку памяти, или занятому. Если обращение выполняется к занятому участку страницы, то запрос удовлетворяется и работа продолжается в штатном режиме. Если обращение выполняется к освобожденному участку памяти, то создается программная исключительная ситуация.

Ссылки, существующие только на стеке

- Передача параметров по ссылке:

```
int a = 10;  
Interlocked.Exchange(ref a, 20); // a: 20
```

- Сохранение ссылки на элемент массива:

```
var array = new int[] { 1, 2, 3 };  
ref int element = ref array[0];  
element = 20; // array: { 20, 2, 3 }
```

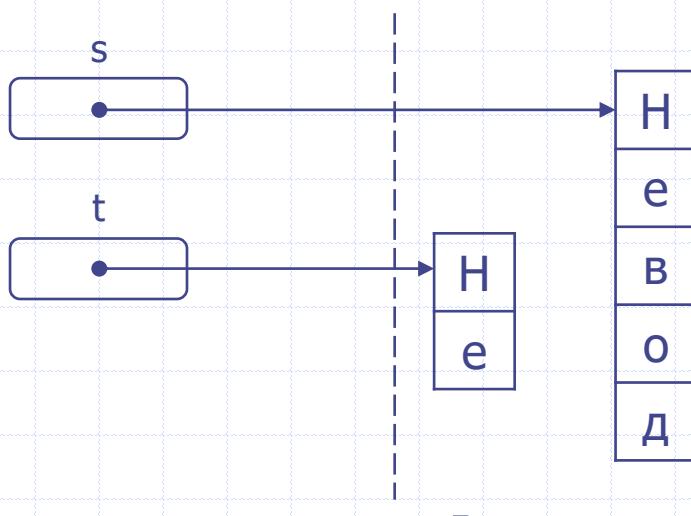
- Сохранение ссылки на поле объекта:

```
var point = new Point { X = 10, Y = 30 };  
ref int x = ref point.X;  
x = 20; // point: { X = 20, Y = 30 }
```

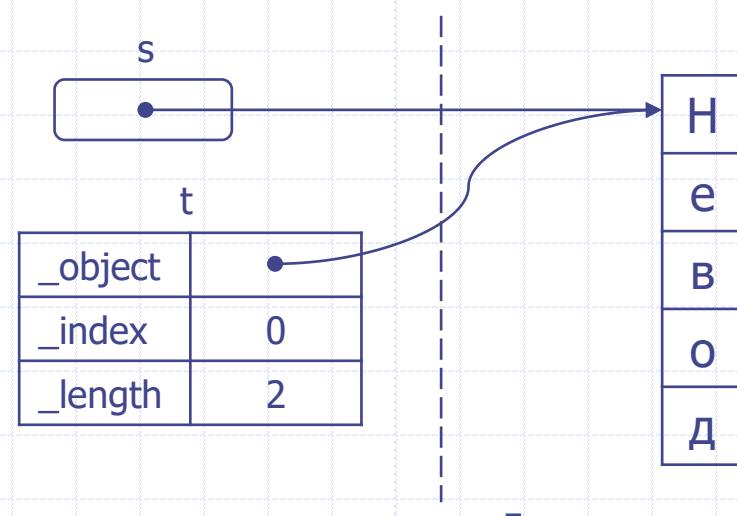
Избежание мусора. Тип данных Memory<T>

- Memory<T> и ReadOnlyMemory<T> позволяют единообразно работать с массивом, строкой или их частью без копирования.

```
string s = "Невод";
string t = s.Substring(0, 2);
```



```
string s = "Невод";
ReadOnlyMemory<char> t = s.AsMemory(0, 2);
```



Тип данных Memory<T>. Пример

- Демонстрирует уменьшение количества выделений памяти при разборе строк:

```
public class Parser
{
    private string text;
    private int pos;

    public Parser(string text) { this.text = text; }

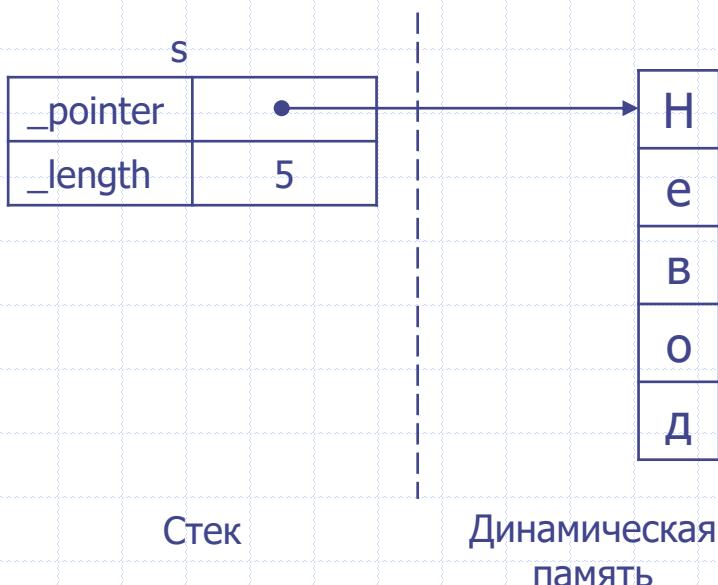
    public string NextTokenAsString()
    {
        int start = pos;
        while (pos < text.Length && text[pos] != ' ')
            pos++;
        return text.Substring(start, pos++ - start); // Создание новой строки
    }

    public ReadOnlyMemory<char> NextTokenAsMemory()
    {
        int start = pos;
        while (pos < text.Length && text[pos] != ' ')
            pos++;
        return text.AsMemory(start, pos++ - start); // Ссылка на фрагмент строки
    }
}
```

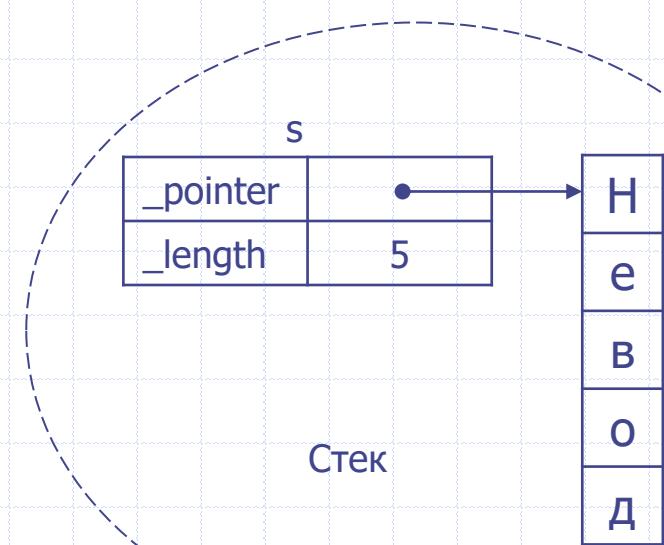
Избежание мусора. Тип данных Span<T>

- Span<T> и ReadOnlySpan<T> позволяют аналогично Memory<T> и ReadOnlyMemory<T> работать не только с массивами в динамической памяти и строками, но и с массивами, выделенными на стеке, а также указателями на неуправляемую память.

```
ReadOnlySpan<char> s = "Невод".AsSpan();
```



```
ReadOnlySpan<char> s = stackalloc  
char[] {'Н', 'е', 'в', 'о', 'д'};
```



Избежание мусора. Оператор stackalloc

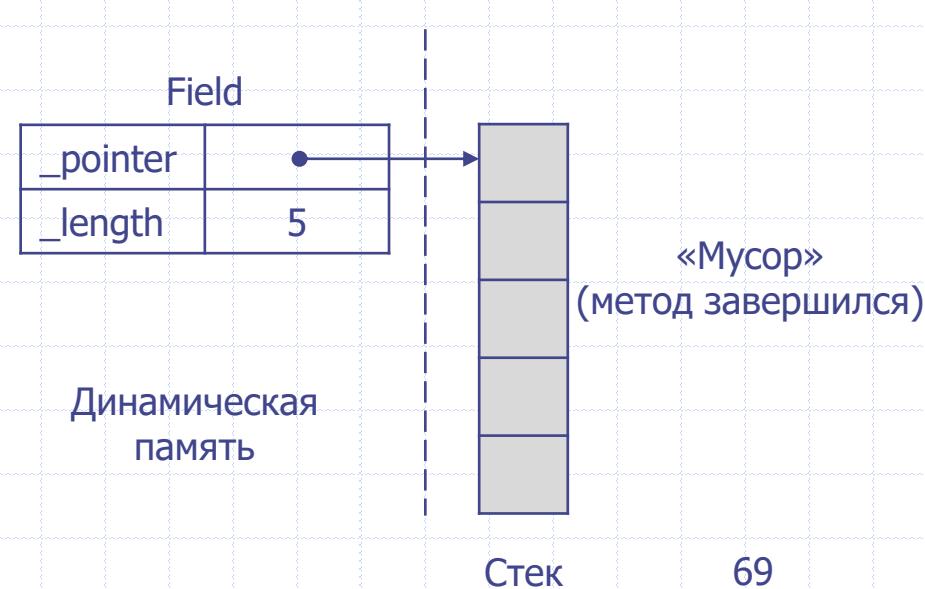
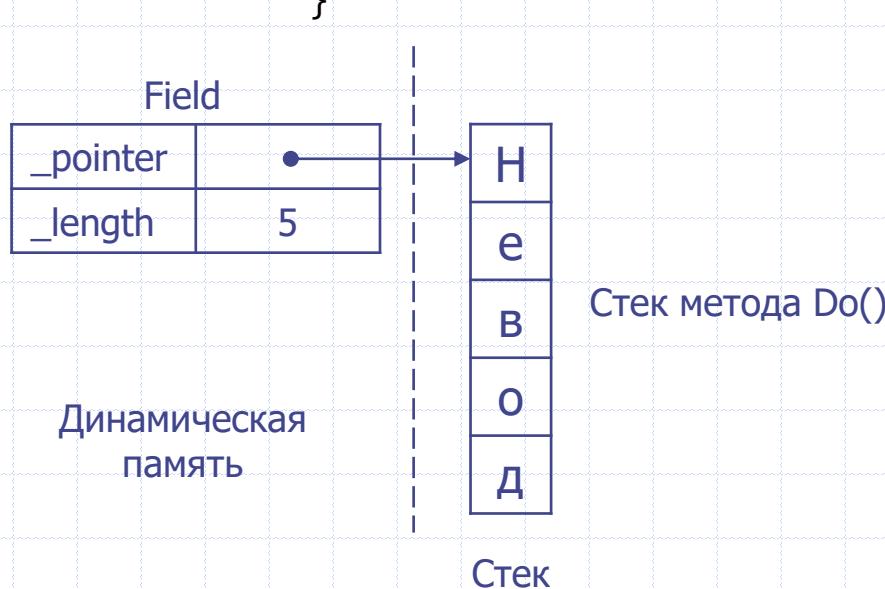
- Выделение массива на стеке используется для оптимизации, если нужно выделить буфер небольшого размера.
- Такой массив не требует сборки мусора, так как автоматически становится свободной памятью после выхода из подпрограммы.

```
public static string IntegerToString(int n)
{
    Span<char> buffer = stackalloc char[16];
    int i = buffer.Length;
    do
    {
        buffer[--i] = (char)(n % 10 + '0');
        n /= 10;
    } while (n != 0);
    return buffer.Slice(i, buffer.Length - i).ToString();
}
```

Тип данных Span<T>. Ограничения

- Из-за возможности указывать на данные в стеке, `Span<T>` и `ReadOnlySpan<T>` нельзя сохранять в динамическую память, иначе после выхода из подпрограммы в динамической памяти останется «зависшая» ссылка.

```
public static class MyClass
{
    public static ReadOnlySpan<char> Field; Запрещено в C#: приведет к ошибке компиляции
    public static void Do()
    {
        Field = stackalloc char[] { 'Н', 'е', 'в', 'о', 'д' }
    }
}
```



«Мусор»
(метод завершился)

Избежание мусора. Типы данных `ref struct`

- Фрагмент реализации `Span<T>`:

```
public readonly ref struct Span<T>
{
    internal ref T _pointer; // Псевдокод: поле-ссылка,
    // аналогичная ссылочному параметру в методе.
    // Недоступна в языке C# или в IL-коде.
    // Реализована на уровне исполняющей среды.
}
```

- Типы, для которых запрещено сохранение в динамическую память, определяются как `ref struct`. Для таких типов компилятором гарантируется отсутствие упаковки и хранения в динамической памяти.
- Необходимость объявить пользовательский тип данных как `ref struct` возникает когда нужно сохранить в поле другой `ref struct`, например `Span<T>` или `ReadOnlySpan<T>`.

Делегаты C# и .NET

- Групповые делегаты – C# (.NET)
- Процедурные переменные – Оберон
- Указатели на методы – Delphi
- Отсутствуют в Java

Пример делегата

```
public delegate void Callback(string text);

class Program
{
    static string[] Commands = { "Ключ на старт", "Протяжка-1", "Продувка",
        "Протяжка-2", "Ключ на дренаж", "Пуск", "Зажигание" };

    static void LogCommands(Callback callback) {
        foreach (string cmd in Commands)
            if (callback != null)
                callback(cmd);
    }

    static void WriteToConsole(string text) {
        Console.WriteLine(text);
    }

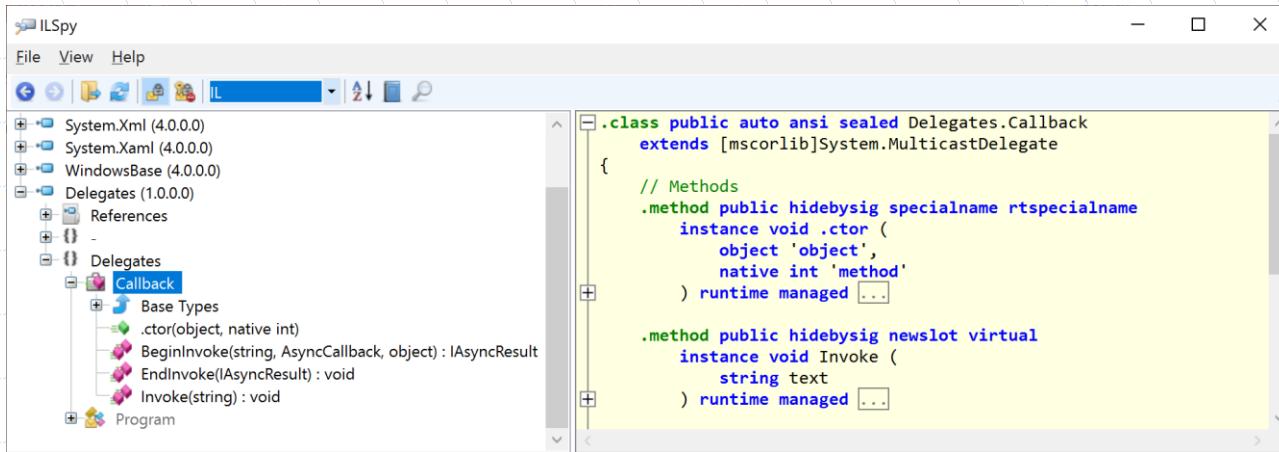
    void WriteToFile(string text) {
        using (var writer = new StreamWriter("log.txt", append: true))
            writer.WriteLine(text);
    }

    static void Main(string[] args) {
        Callback staticCallback = new Callback(WriteToConsole);
        Program p = new Program();
        Callback instanceCallback = new Callback(p.WriteToFile);

        LogCommands(staticCallback);
        LogCommands(instanceCallback);
    }
}
```

Тип данных delegate это класс

```
public delegate void Callback(string text);  
  
// Компилятор определяет класс:  
  
public class Callback : MulticastDelegate  
{  
    public extern Callback(object @object, IntPtr method);  
  
    public virtual extern IAsyncResult BeginInvoke(  
        string text, AsyncCallback callback, object @object);  
  
    public virtual extern void EndInvoke(IAsyncResult result);  
  
    public virtual extern void Invoke(string text);  
}
```



Тип данных Delegate

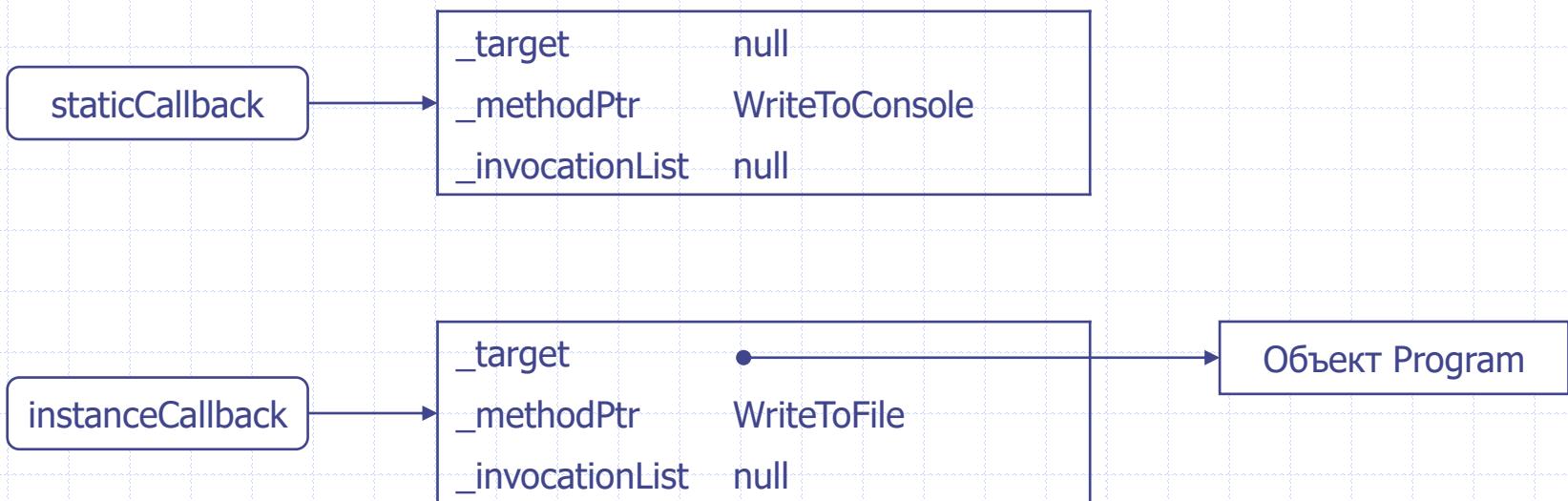
```
public abstract class Delegate : ICloneable, ISerializable
{
    public MethodInfo Method { get; }
    public object Target { get; }

    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);

    public static Delegate CreateDelegate(Type type, object target, string method,
        bool ignoreCase);
    public static Delegate CreateDelegate(Type type, MethodInfo method);
    public static Delegate CreateDelegate(Type type, object firstArgument,
        MethodInfo method, bool throwOnBindFailure);
    public static Delegate CreateDelegate(Type type, object firstArgument,
        MethodInfo method);
    public static Delegate CreateDelegate(Type type, MethodInfo method,
        bool throwOnBindFailure);
    public static Delegate CreateDelegate(Type type, Type target, string method,
        bool ignoreCase, bool throwOnBindFailure);
    public static Delegate CreateDelegate(Type type, Type target, string method,
        bool ignoreCase);
    public static Delegate CreateDelegate(Type type, object target, string method);
    public static Delegate CreateDelegate(Type type, Type target, string method);
    public static Delegate CreateDelegate(Type type, object target, string method,
        bool ignoreCase, bool throwOnBindFailure);

    public object DynamicInvoke(params object[] args);
    public virtual Delegate[] GetInvocationList();
}
```

```
Callback staticCallback = new Callback(WriteToConsole);  
Program p = new Program();  
Callback instanceCallback = new Callback(p.WriteToFile);
```



Пример группового делегата

```
public delegate void Callback(string text);

class Program
{
    static string[] Commands = { "Ключ на старт", "Протяжка-1", "Продувка",
        "Протяжка-2", "Ключ на дренаж", "Пуск", "Зажигание" };

    static void LogCommands(Callback callback)
    {
        foreach (string cmd in Commands)
            if (callback != null)
                callback(cmd);
    }

    static void WriteToConsole(string text)
    {
        Console.WriteLine(text);
    }

    static void WriteToMessageBox(string text)
    {
        MessageBox.Show(text);
    }

    void WriteToFile(string text)
    {
        using (var writer = new StreamWriter(
            "log.txt", append: true))
            writer.WriteLine(text);
    }
}

static void Main(string[] args) {
    Program p = new Program();

    Callback chain = null;
    Callback cb1 = WriteToConsole;
    Callback cb2 = WriteToMessageBox;
    Callback cb3 = p.WriteLine;

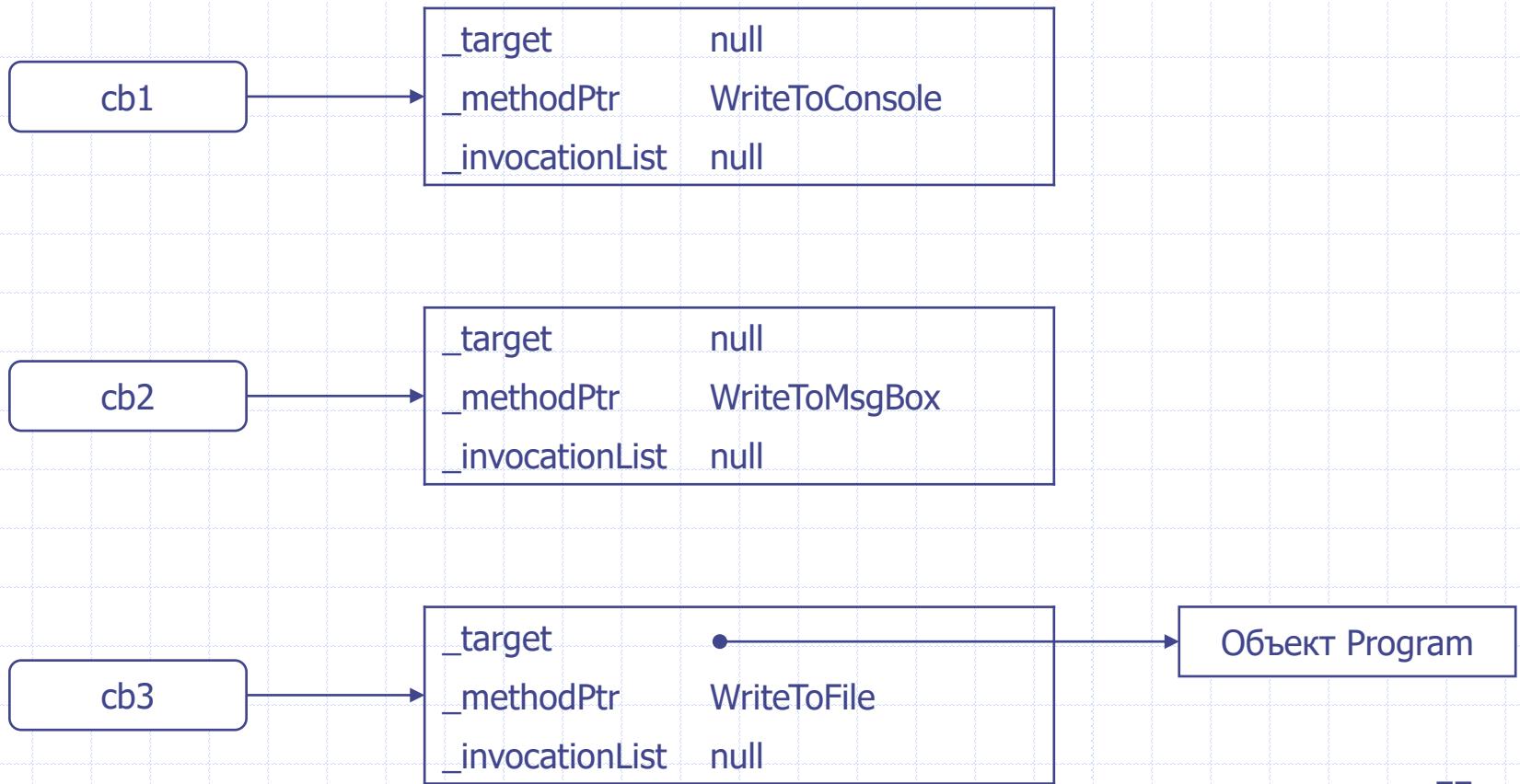
    chain += cb1;
    chain += cb2;
    chain += cb3;

    LogCommands(chain);

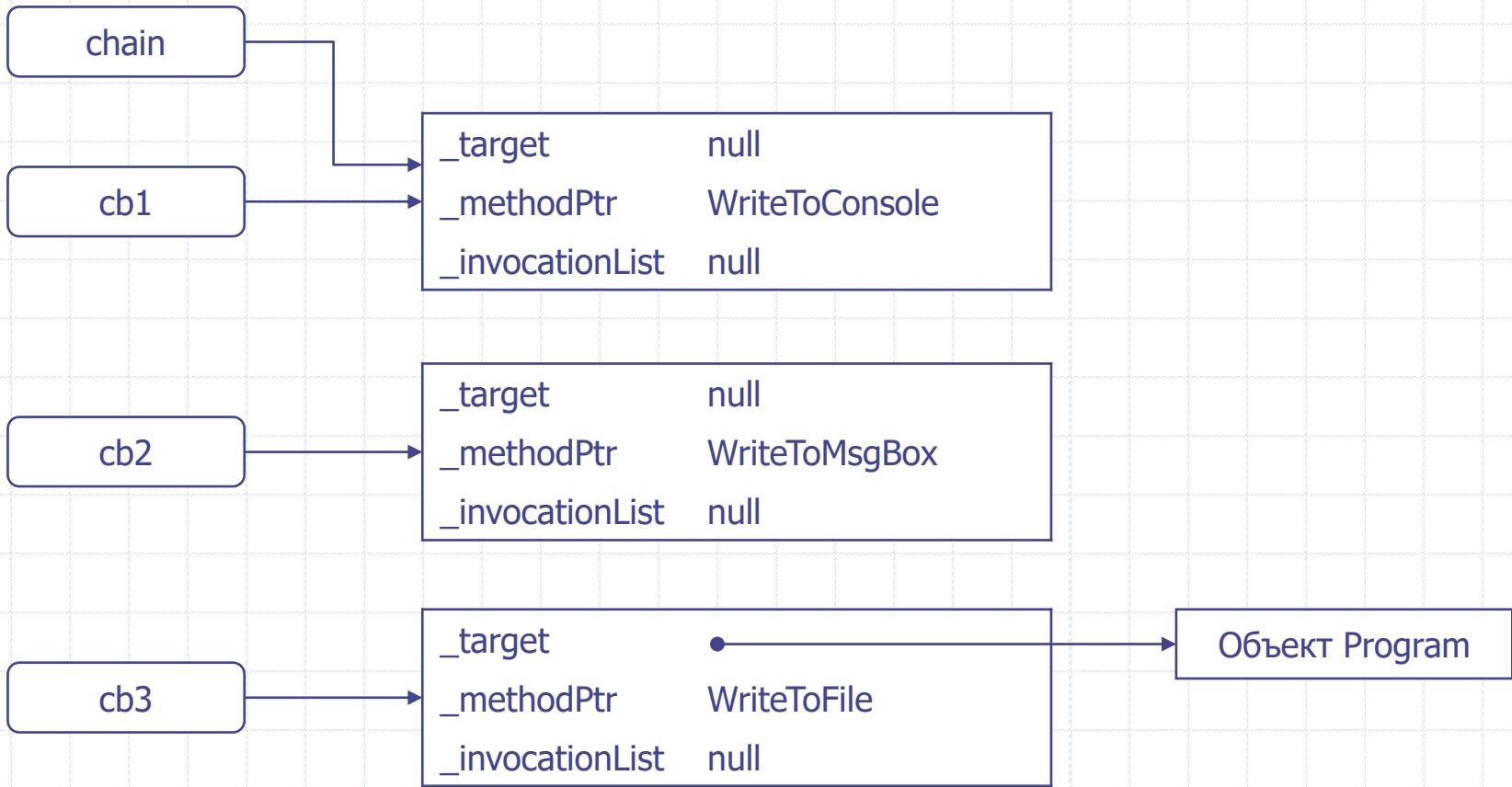
    chain -= WriteToMessageBox;

    LogCommands(chain);
}
```

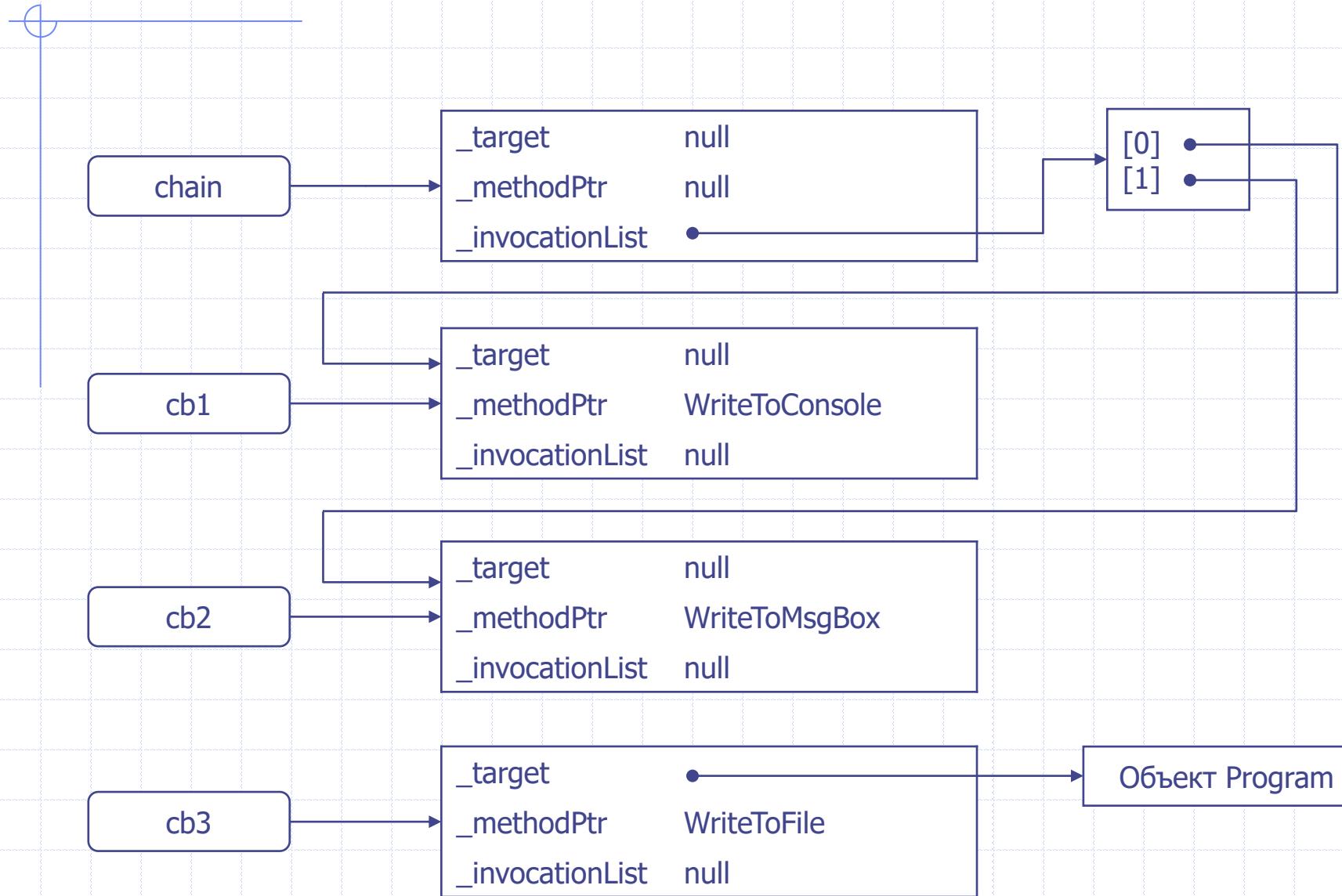
Callback cb1 = WriteToConsole;
Callback cb2 = WriteToMsgBox;
Callback cb3 = p.WriteLine;



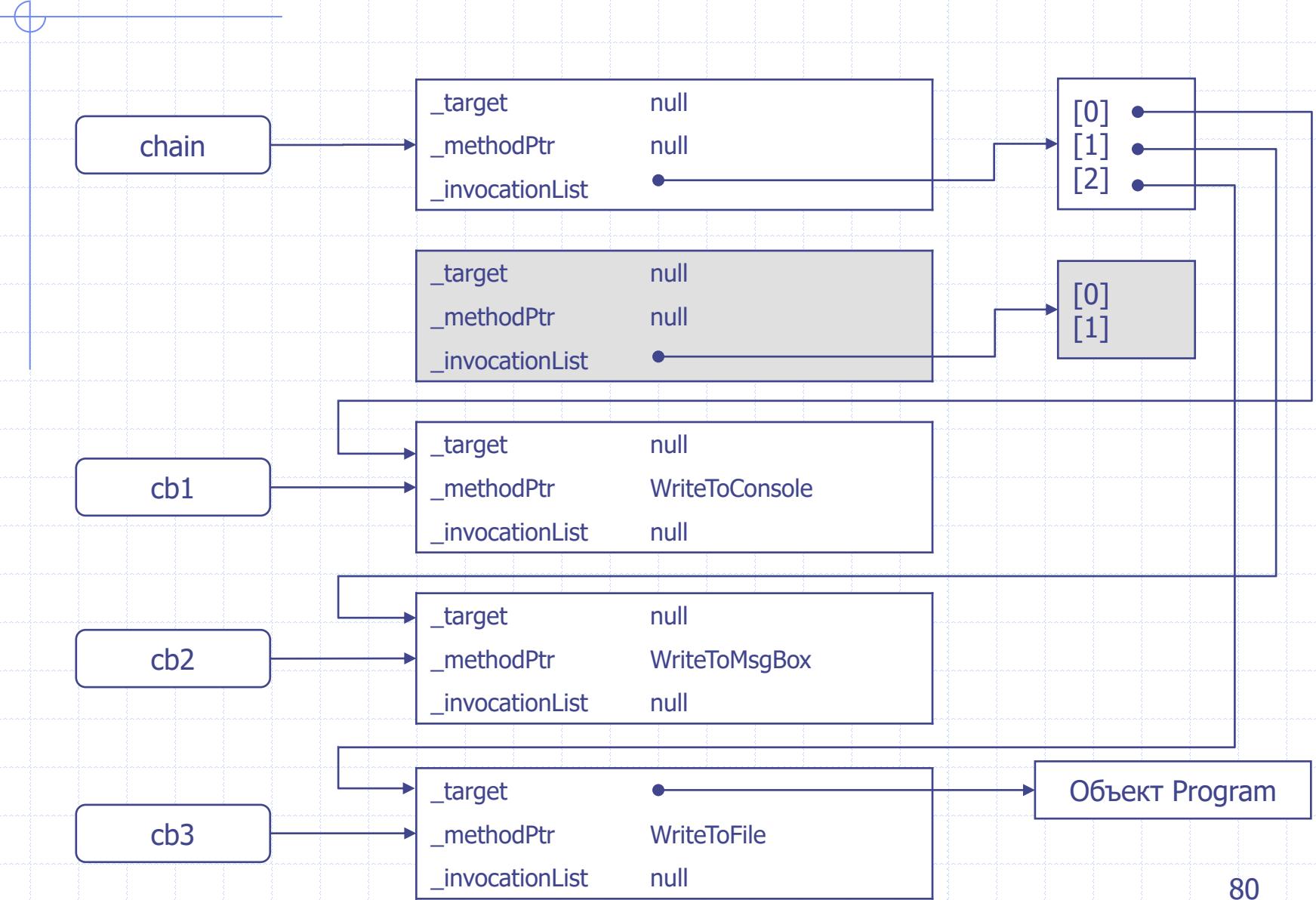
chain += cb1;



chain += cb2;



chain += cb3;



Метод Invoke в классе делегата (псевдо-код)

```
public class Callback : MulticastDelegate
{
    public extern Callback(object @object, IntPtr method);

    public virtual void Invoke(string text)
    {
        // Псевдо-код, создаётся динамической компиляцией
        Delegate[] delegates = _invocationList as Delegate[];
        if (delegates != null)
        {
            foreach (Delegate d in delegates)
            {
                d.Invoke(text);
            }
        }
        else
        {
            if (_target != null)
                _target._methodPtr(text); // псевдо-код
            else
                _methodPtr(text); // псевдо-код
        }
    }
}
```

Использование метода GetInvocationList для безопасного вызова группы делегатов

```
public static void SafeInvoke(Callback callback, string str)
{
    if (callback != null)
    {
        Delegate[] delegates = callback.GetInvocationList();
        List<Exception> errors = null;
        foreach (Callback d in delegates)
        {
            try
            {
                d(str);
            }
            catch (Exception ex)
            {
                if (errors == null)
                    errors = new List<Exception>();
                errors.Add(ex);
            }
        }
        if (errors != null)
            throw new AggregateException(errors);
    }
}
```

События на основе делегатов

```
public delegate void ListUpdateEvent(ObservableList sender);

public class ObservableList
{
    private ListUpdateEvent updated;
    public List<string> List;

    public event ListUpdateEvent Updated {
        add { updated += value; }
        remove { updated -= value; }
    }

    public ObservableList(List<string> list) {
        List = list;
    }

    public void Add(string item) {
        List.Add(item);
        if (updated != null) updated(this);
    }

    public bool Remove(string item) {
        var result = List.Remove(item);
        if (updated != null) updated(this);
        return result;
    }
}
```

```
class Program
{
    static void HandleEvent(
        ObservableList sender)
    {
        Console.WriteLine(
            "Количество элементов: {0}",
            sender.List.Count);
    }

    static void Main(string[] args)
    {
        var list = new ObservableList(
            new List<string>());

        list.Updated += HandleEvent;
        // Запрещено:
        // list.Updated = HandleEvent;
        // list.Updated(list);

        list.Add("Ключ на старт");
        list.Add("Протяжка-1");

        list.Updated -= HandleEvent;
        list.Add("Продувка");
    }
}
```

Отличия события от делегата

- Событие это виртуальное свойство-делегат, для которого можно переопределить методы добавления и удаления обработчиков (операторы += и -=).

```
public event ListUpdateEvent Updated
{
    add { updated += value; }
    remove { updated -= value; }
}
list.Updated += HandleEvent;
```

- Запрещено присваивание значений событию (оператор =) снаружи объекта.

```
list.Updated = HandleEvent; // Запрещено
```

- Запрещено вызывать обработчики события (метод Invoke) снаружи объекта.

```
list.Updated(list); // Запрещено
```

Анонимные делегаты языка C#

```
class Program
{
    static string[] Commands = { "Ключ на старт", "Протяжка-1", "Продувка",
        "Протяжка-2", "Ключ на дренаж", "Пуск", "Зажигание" };

    public static bool StaticPredicate(string arg)
    {
        return arg.StartsWith("Ключ");
    }

    static void Main(string[] args)
    {
        string prefix = "Ключ";
        System.Func<string, bool> inPlacePredicate = delegate(string x)
        {
            return x.StartsWith(prefix);
        };

        var subset = Commands.Where(StaticPredicate);
        subset = Commands.Where(inPlacePredicate);
        subset = Commands.Where(
            delegate (string x)
            {
                return x.StartsWith(prefix);
            }
        );
        subset = Commands.Where((x) => { return x.StartsWith(prefix); });
        subset = Commands.Where(x => x.StartsWith(prefix));

        Console.WriteLine(string.Join("\n", subset));
    }
}
```

Многопоточное программирование

- Модель многопоточности .NET соответствует обобщенной модели целевых операционных систем:
 - Адресное пространство процесса общее для всех потоков.
 - Отдельный стек для каждого потока.
 - Вытесняющая многозадачность.
 - Единицей диспетчеризации является поток.
- Для создания самостоятельных потоков и управления заданными потоком используется класс `Thread`.
- Для выполнения подпрограмм в контексте готовых потоков, создаваемых исполняющей средой в пуле потоков, используется класс `ThreadPool`.

Создание самостоятельного потока. Класс Thread

```
class Program
{
    public static void Main()
    {
        Thread t1 = new Thread(DoWork);
        t1.Start("X");
        Thread t2 = new Thread(DoWork, maxStackSize: 16 * 1024);
        t2.Start("o");
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nПоток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

Создание самостоятельного потока.

Класс Thread

```
public sealed class Thread : CriticalFinalizerObject
{
    public Thread(ParameterizedThreadStart start);
    public Thread(ThreadStart start);
    public Thread(ParameterizedThreadStart start, int maxStackSize);
    public Thread(ThreadStart start, int maxStackSize);
    ~Thread();
    public static Thread CurrentThread { get; }
    public CultureInfo CurrentCulture { get; set; }
    public CultureInfo CurrentUICulture { get; set; }
    public bool IsAlive { get; }
    public bool IsBackground { get; set; }
    public bool IsThreadPoolThread { get; }
    public int ManagedThreadId { get; }
    public string Name { get; set; }
    public ExecutionContext ExecutionContext { get; }
    public ThreadPriority Priority { get; set; }
    public ThreadState ThreadState { get; }
    public static int GetCurrentProcessorId();
    public static void Sleep(int millisecondsTimeout);
    public static void Sleep(TimeSpan timeout);
    public static bool Yield();
    public void Interrupt();
    public bool Join(TimeSpan timeout);
    public bool Join(int millisecondsTimeout);
    public void Join();
    public void Start(object parameter);
    public void Start();
    ...
}
```

Фоновые потоки платформы .NET

- Фоновые (background) потоки отличаются от основных (foreground) тем, что завершаются (прерываются) автоматически при завершении всех основных потоков.
- Главный поток является основным. Если он завершается, то для завершения программы необходимо, чтобы все остальные основные потоки тоже завершились.

Создание самостоятельного фонового потока

```
class Program
{
    public static void Main()
    {
        Thread t1 = new Thread(DoWork) { IsBackground = true };
        t1.Start("X");
        Thread t2 = new Thread(DoWork) { IsBackground = true };
        t2.Start("o");
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nПоток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

Выполнение подпрограмм с помощью пула потоков. Класс ThreadPool

```
class Program
{
    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(DoWork, "o");
        ThreadPool.QueueUserWorkItem(DoWork, "X");
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nРаботает поток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

Выполнение любого количества задач с помощью пула потоков

```
class Program
{
    public static void Main()
    {
        for (int i = 0; i < 10000; i++)
            ThreadPool.QueueUserWorkItem(DoWork, i);
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nРаботает поток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

Настройки пула потоков

```
class Program
{
    public static void Main()
    {
        ThreadPool.GetMinThreads(out int minThreads, out int minIO);
        Console.WriteLine($"Min Threads = {minThreads}");
        ThreadPool.GetMaxThreads(out int maxThreads, out int maxIO);
        Console.WriteLine($"Max Threads = {maxThreads}");

        for (int i = 0; i < 10000; i++)
            ThreadPool.QueueUserWorkItem(DoWork, i);

        for (int i = 0; i < 1000; ++i)
        {
            ThreadPool.GetAvailableThreads(out int threads, out int IO);
            Console.WriteLine($"{Environment.NewLine}Active Threads = {maxThreads - threads}");
            Thread.Sleep(100);
        }
        Console.ReadLine();
    }
}
```

Почему по умолчанию максимальное число потоков в пуле потоков такое большое?

- Ограничение на максимальное количество потоков может приводить к бесконечной блокировке.
- Если максимальное количество потоков в пуле равно, например, 10 и все 10 задач (делегатов) переходят в ожидание, то это ожидание будет бесконечным, потому что больше нет потока, который мог бы изменить ожидаемые условия.

Бесконечная блокировка из-за исчерпания пула потоков. Пример

```
class Program {
    static volatile int CompletedTaskNumber = 0;
    static void Main(string[] args) {
        ThreadPool.GetMinThreads(out int workerThreads, out _);
        ThreadPool.SetMaxThreads(workerThreads, workerThreads);
        int n = workerThreads + 1; // если n = workerThreads, то нет блокировки
        ThreadPool.QueueUserWorkItem(Do, n);
        while (CompletedTaskNumber != n) Thread.Yield();
        Console.ReadLine();
    }
    static void Do(object value) {
        int current = (int) value;
        if (current > 1) {
            int next = current - 1;
            Console.WriteLine($"Задача {current} ожидает задачу {next}");
            ThreadPool.QueueUserWorkItem(Do, next);
            while (CompletedTaskNumber != next) Thread.Yield();
        }
        Console.WriteLine($"Задача {current} завершается");
        CompletedTaskNumber = current;
    }
}
```

Проблема синхронизации потоков при работе с общими данными

```
public class CountedObject
{
    public static int count = 0;

    public CountedObject()
    {
        count = count + 1;
    }

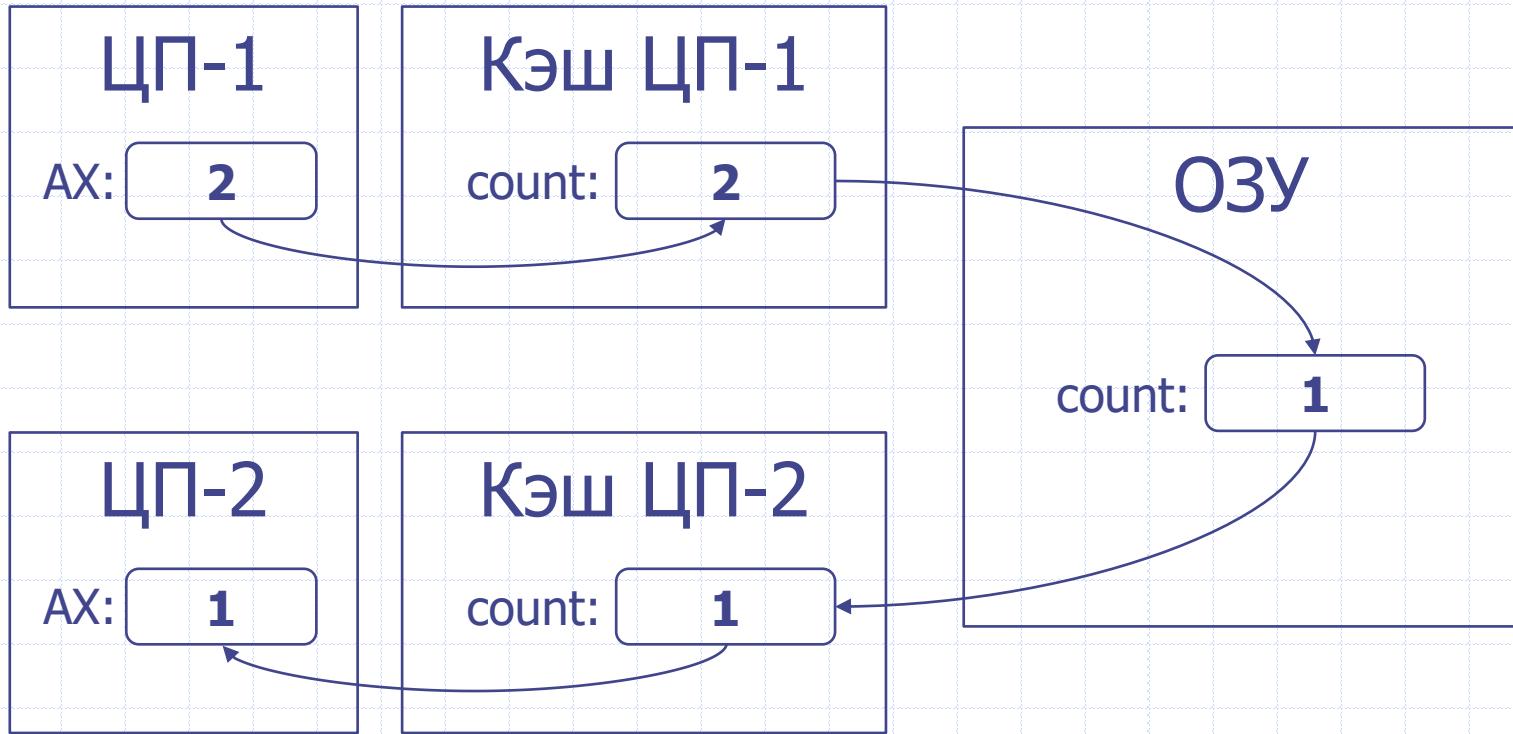
    ~CountedObject()
    {
        count -= 1;
    }
}
```

```
ldsfld    int32 CountedObject::count
ldc.i4.1
add
stsfld    int32 CountedObject::count
-----  

mov AX, [count]
inc AX
mov [count], AX
```

Поток 1	Поток 2	Результат
mov AX, [count]		AX ₁ = 0, count = 0
	mov AX, [count]	AX ₂ = 0, count = 0
	inc AX	AX ₂ = 1, count = 0
	mov [count], AX	AX ₂ = 1, count = 1
inc AX		AX ₁ = 1, count = 1
mov [count], AX		AX ₁ = 1, count = 1

Проблема синхронизации потоков при работе с общими данными на многопроцессорной системе



Синхронизация с помощью мьютекса. Оператор lock

```
public class CountedObject
{
    public static int count = 0;
    static object sync = new object();

    public CountedObject()
    {
        lock (sync)
            count += 1;
    }

    ~CountedObject()
    {
        lock (sync)
            count -= 1;
    }
}
```

```
    Monitor.Enter(sync);
    try
    {
        count += 1;
    }
    finally
    {
        Monitor.Exit(sync);
    }
```

Синхронизация с помощью мьютекса. Оператор lock

```
public class CountedObject
{
    public static int count = 0;
    static object sync = new object();

    public CountedObject()
    {
        lock (sync)
            count += 1;
    }

    ~CountedObject()
    {
        lock (sync)
            count -= 1;
    }
}

bool lockTaken = false;
try
{
    Monitor.Enter(sync, ref lockTaken);
    count += 1;
}
finally
{
    if (lockTaken)
        Monitor.Exit(sync);
}
```

Синхронизация с помощью атомарных операций. Класс Interlocked

```
public class CountedObject
{
    public static int count = 0;

    public CountedObject()
    {
        Interlocked.Increment(ref count);
    }

    ~CountedObject()
    {
        Interlocked.Decrement(ref count);
    }
}
```

Класс Interlocked

```
public static class Interlocked
{
    public static int Increment(ref int location);
    public static int Decrement(ref int location);

    public static extern int Exchange(
        ref int location1, int value);

    public static extern object Exchange(
        ref object location1, object value);

    public static extern int CompareExchange(
        ref int location1, int value, int comparand);

    public static extern object CompareExchange(
        ref object location1, object value, object comparand);
}
```

Класс Interlocked. Логика работы метода CompareExchange

```
public static class Interlocked
{
    // Псевдокод, выполняется атомарно процессором
    public static int CompareExchange(
        ref int location, int value, int comparand)
    {
        // Записать в location значение из value при условии,
        // что в location было значение comparand.
        // Вернуть старое значение location.
        int result = location;
        if (location == comparand)
            location = value;
        return result;
    }
}
```

Реализация мьютекса с помощью Interlocked.CompareExchange

```
public class Mutex
{
    private Thread thread;

    public void Lock()
    {
        Thread t = Thread.CurrentThread;
        while (Interlocked.CompareExchange(ref thread, t, null) != null)
            Thread.Yield();
        Thread.MemoryBarrier();
    }

    public void Unlock()
    {
        Thread t = Thread.CurrentThread;
        if (Interlocked.CompareExchange(ref thread, null, t) != t)
            throw new SynchronizationLockException();
        Thread.MemoryBarrier();
    }
}
```

Сравнение способов синхронизации

Interlocked-операция	Объект с блокировкой оператором lock
Синхронизация доступа к одной переменной	Синхронизация доступа к множеству переменных (к любому количеству данных)
Согласование между всеми процессорами одной строки кэша , в которую попала переменная	Согласование между всеми процессорами всех строк кэша со всеми изменёнными значениями
Реализация в аппаратуре на уровне процессора	Реализация через две Interlocked-операции (для обеспечения рекурсивности) и объект ожидания ОС , используемый после большого количества неудачных попыток войти в блокировку
Стоимость мала	Стоимость мала в оптимистическом сценарии и высока в пессимистическом сценарии

Переменные с модификатором volatile

```
class Program
{
    static volatile bool stopped = false;

    static void Main(string[] args)
    {
        var thread = new Thread(DoLongWork);
        thread.Start();
        Console.WriteLine("Работа начата");
        Thread.Sleep(500);
        stopped = true;
        thread.Join();
        Console.WriteLine("Работа завершена");
    }

    static void DoLongWork(object state)
    {
        bool work = false;
        while (!stopped)
            work = !work;
    }
}
```

Мониторы Хоара. Класс Monitor

- Мониторы Хоара позволяют реализовать наиболее сложные сценарии синхронизации потоков при доступе к данным с ожиданием условий.
- Класс Monitor предоставляет четыре операции:
 - Enter – вход в блокировку;
 - Exit – выход из блокировки;
 - Wait – ожидание сигнала;
 - Pulse/PulseAll – отправка сигнала.
- Операции Wait, Pulse/PulseAll требуют предварительного входа в блокировку.

Запуск и ожидание выполнения массива делегатов с помощью класса Monitor

```
public class ActionRunner
{
    int runningCount;
    object sync = new object();

    public void RunAndWaitAll(Action[] actions)
    {
        runningCount = actions.Length;
        foreach (Action action in actions)
            ThreadPool.QueueUserWorkItem(ExecuteAction, action);
        lock (sync)
            if (runningCount > 0)
                Monitor.Wait(sync);
    }

    private void ExecuteAction(object state)
    {
        var action = (Action)state;
        action();
        lock (sync)
        {
            runningCount--;
            if (runningCount == 0)
                Monitor.Pulse(sync);
        }
    }
}
```

// Monitor.Exit(sync);
// Ожидание();
// Monitor.Enter(sync);

Реализация собственного пула потоков с помощью класса Monitor

```
public class TaskQueue
{
    private List<Thread> threads;
    private Queue<Action> tasks;

    public TaskQueue(int threadCount)
    {
        tasks = new Queue<Action>();
        threads = new List<Thread>();
        for (int i = 0; i < threadCount; i++)
        {
            var t = new Thread(DoThreadWork);
            threads.Add(t);
            t.IsBackground = true;
            t.Start();
        }
    }

    public void EnqueueTask(Action task)
    {
        lock (tasks)
        {
            tasks.Enqueue(task);
            Monitor.Pulse(tasks);
        }
    }

    private Action DequeueTask()
    {
        lock (tasks)
        {
            while (tasks.Count == 0)
                Monitor.Wait(tasks);
            return tasks.Dequeue();
        }
    }

    private void DoThreadWork()
    {
        while (true)
        {
            Action task = DequeueTask();
            try
            {
                task();
            }
            catch (ThreadAbortException)
            {
                Thread.ResetAbort();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Реализация собственного пула потоков с помощью класса Monitor. Вариант с методом Close

```
public class TaskQueue
{
    private List<Thread> threads;
    private Queue<Action> tasks;

    public TaskQueue(int threadCount)
    {
        tasks = new Queue<Action>();
        threads = new List<Thread>();
        for (int i = 0; i < threadCount; i++)
        {
            var t = new Thread(DoThreadWork);
            threads.Add(t);
            t.IsBackground = true;
            t.Start();
        }
    }

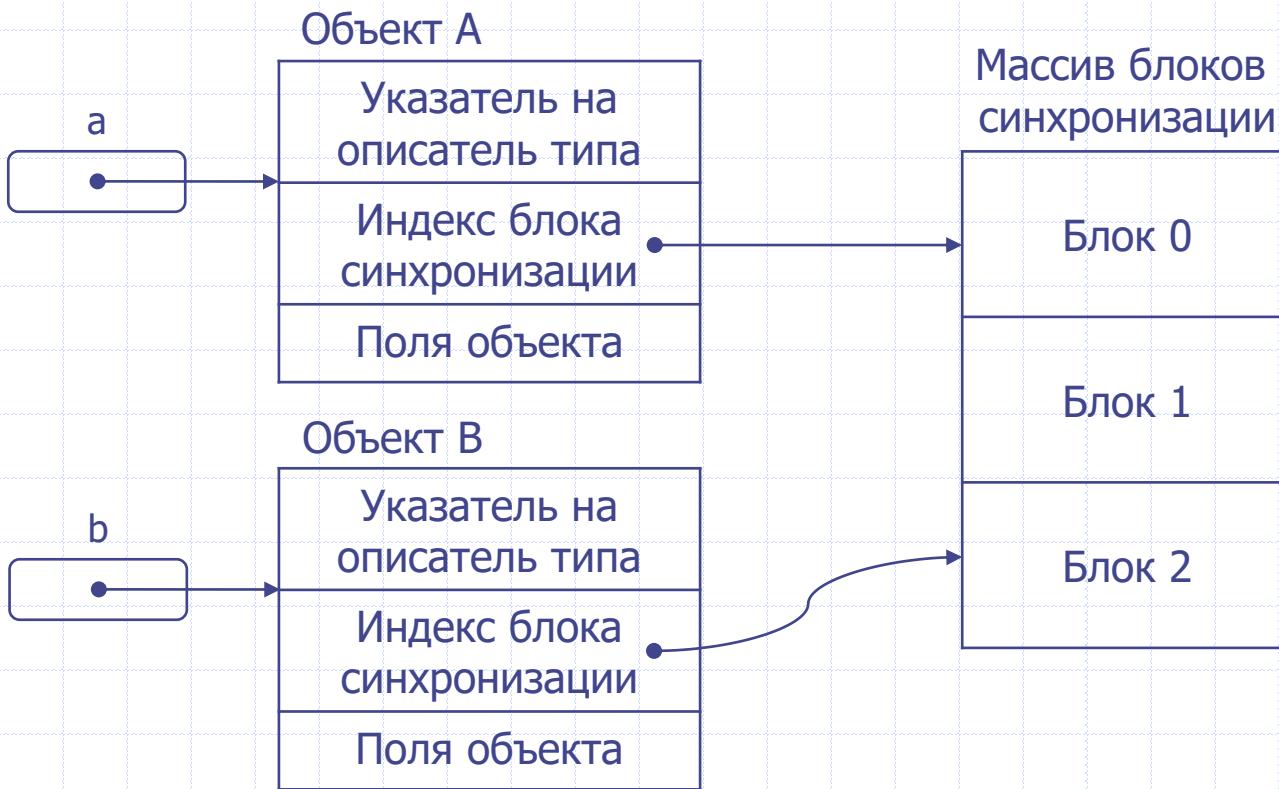
    public void EnqueueTask(Action task)
    {
        lock (tasks)
        {
            tasks.Enqueue(task);
            Monitor.Pulse(tasks);
        }
    }

    public void Close()
    {
        for (int i = 0; i < threads.Count; i++)
            EnqueueTask(null);
        foreach (Thread t in threads)
            t.Join();
    }
}
```

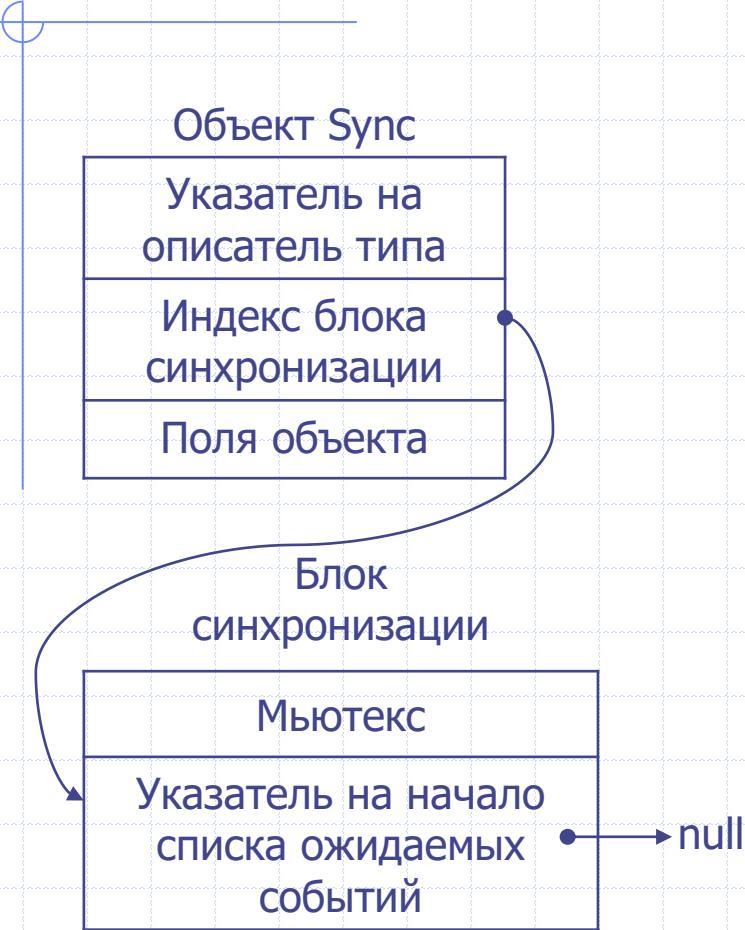
```
private Action DequeueTask()
{
    lock (tasks)
    {
        while (tasks.Count == 0)
            Monitor.Wait(tasks);
        return tasks.Dequeue();
    }
}

private void DoThreadWork()
{
    while (true)
    {
        Action task = DequeueTask();
        if (task != null)
        {
            try {
                task();
            }
            catch (ThreadAbortException) {
                Thread.ResetAbort();
            }
            catch (Exception ex) {
                Console.WriteLine(ex);
            }
        }
        else
            break;
    }
}
```

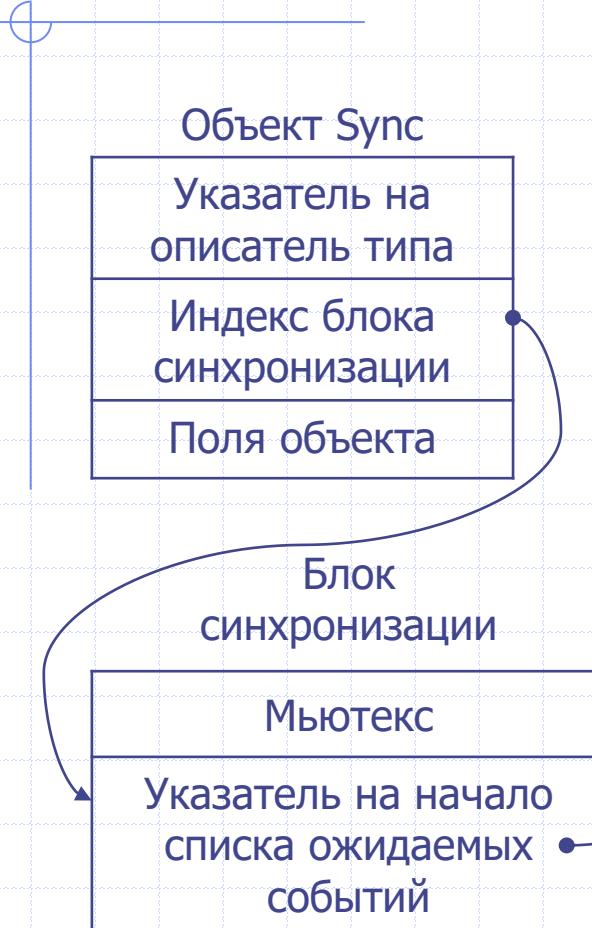
Структура заголовка объекта с индексом блока синхронизации (sync block index)



Реализация методов Wait и Pulse/PulseAll в классе Monitor



Реализация методов Wait и Pulse/PulseAll в классе Monitor

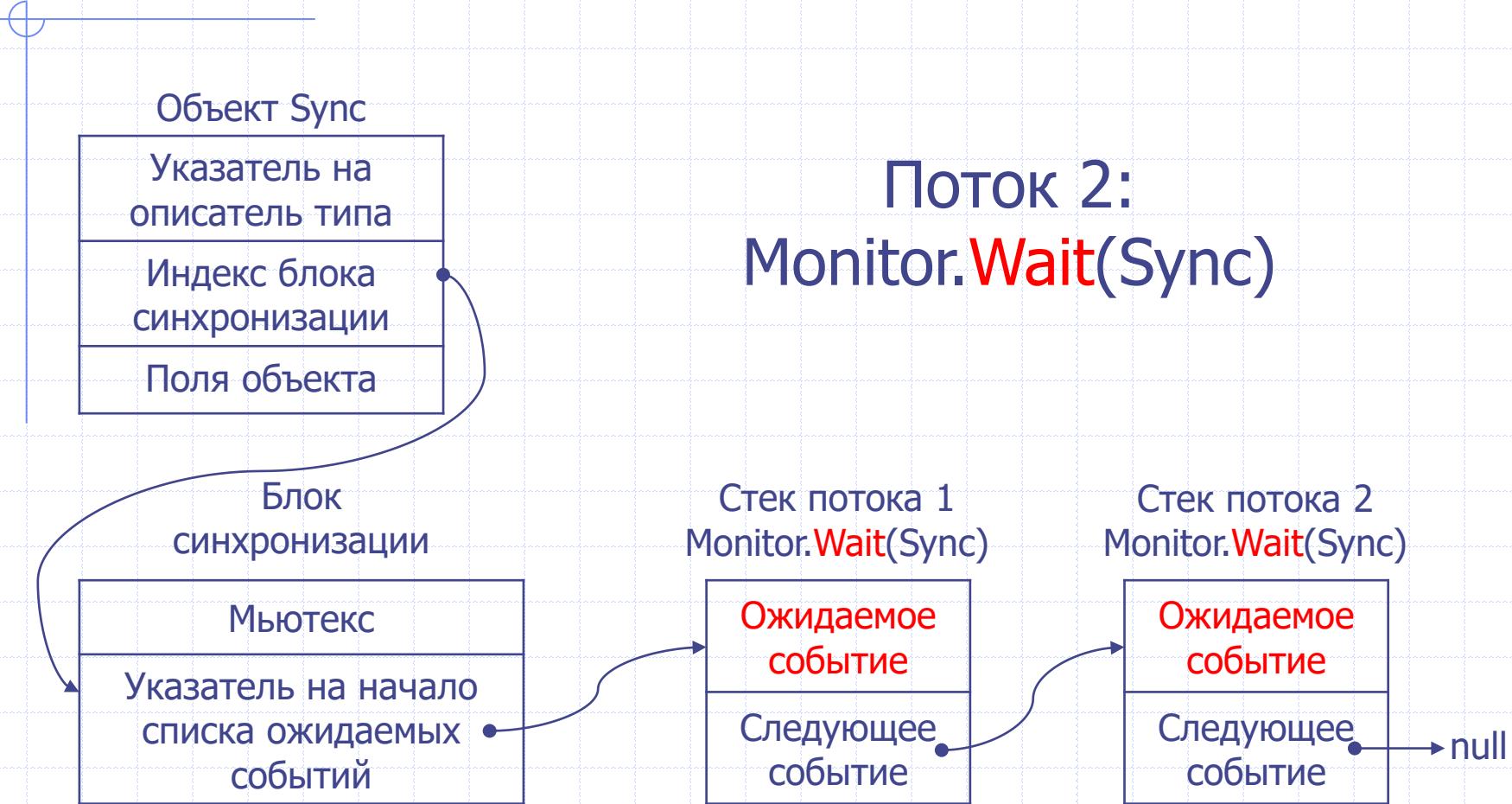


Поток 1:
Monitor.Wait(Sync)

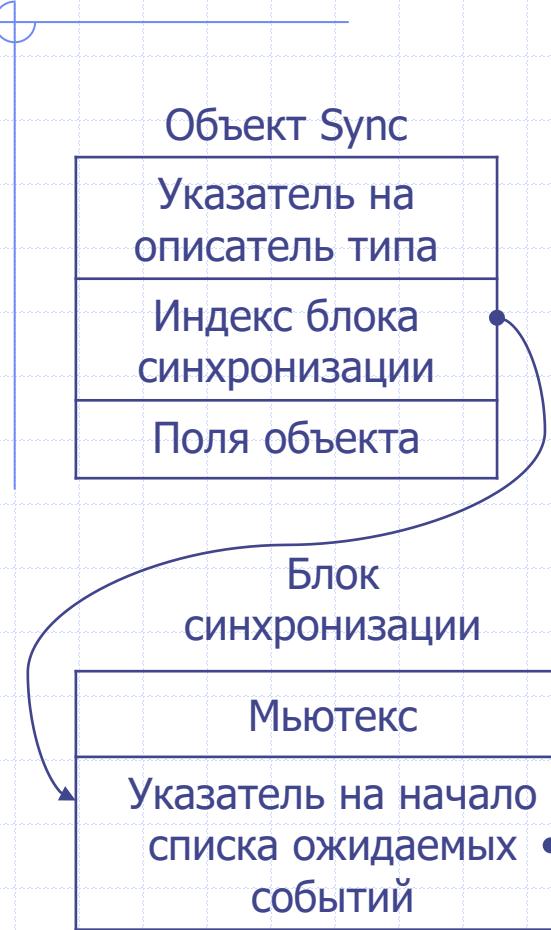
Стек потока 1
Monitor.Wait(Sync)



Реализация методов Wait и Pulse/PulseAll в классе Monitor

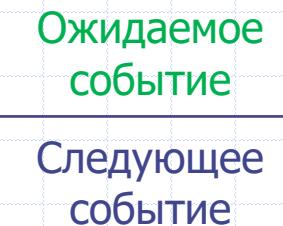


Реализация методов Wait и Pulse/PulseAll в классе Monitor



Поток 3:
Monitor.Pulse(Sync)

Стек потока 1
Monitor.Wait(Sync)

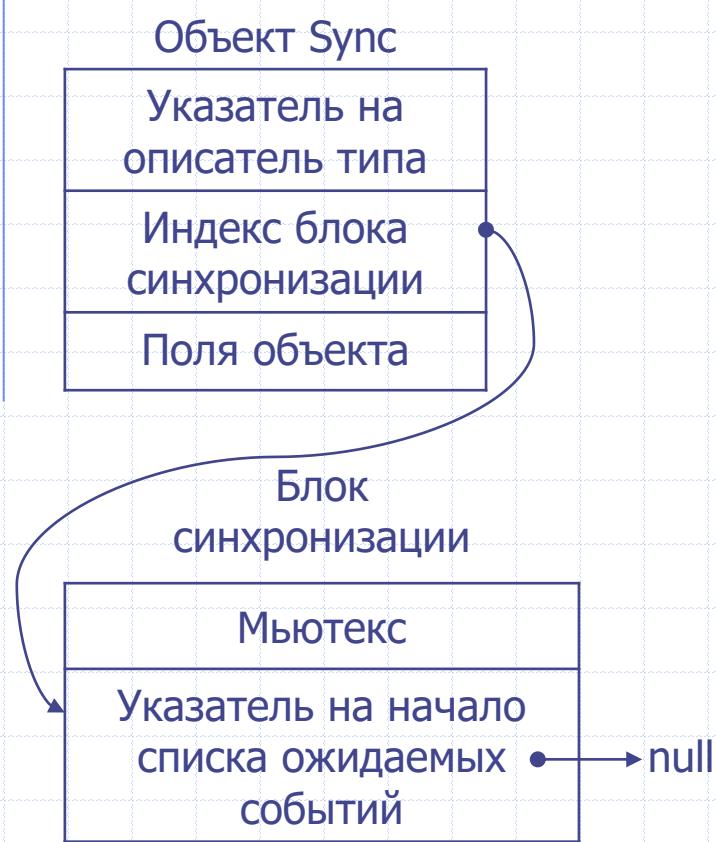


Стек потока 2
Monitor.Wait(Sync)



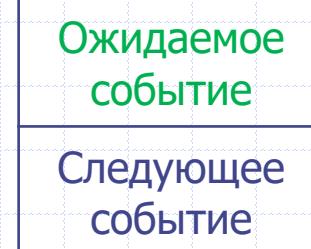
Реализация методов Wait и Pulse/PulseAll в классе Monitor

+

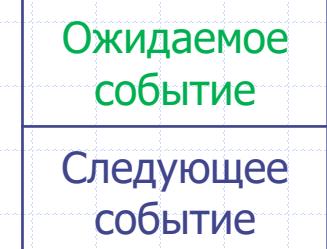


Поток 3:
Monitor.PulseAll(Sync)

Стек потока 1
Monitor.Wait(Sync)



Стек потока 2
Monitor.Wait(Sync)



Локальные данные потока (Thread-Local Storage – TLS)

- Каждый поток может иметь свои выделенные данные, которые изолированы от других потоков.
- Поддержку локальных данных потока обеспечивает операционная система.
- Для каждого потока создается блок данных, ссылка на который помещается в заголовок потока.
- Адрес блока для текущего потока обычно хранится в специальном регистре процессора. При переключении потоков операционная система обновляет значение регистра.
- В операционной системе Windows для процессоров x86-64 для хранения адреса локальных данных текущего потока используются специальные регистры FS и GS.

Локальные данные потока.

Атрибут [ThreadStatic]

```
class Logger
{
    [ThreadStatic]
    public static string TaskName;

    public static void WriteLine(string text)
    {
        Console.WriteLine($"{TaskName}: {text}");
    }
}

class Program
{
    static void Main()
    {
        ThreadPool.QueueUserWorkItem(Worker, "Задача 1");
        ThreadPool.QueueUserWorkItem(Worker, "Задача 2");
        Console.ReadKey();
    }

    static void Worker(object state)
    {
        Logger.TaskName = (string)state;
        for (int i = 0; i < 100; i++)
        {
            Logger.WriteLine("Идёт работа");
            Thread.Sleep(100);
        }
    }
}
```

Локальные данные потока.

Тип данных ThreadLocal<T>

```
class Logger
{
    static readonly ThreadLocal<string> taskName;

    static Logger()
    {
        taskName = new ThreadLocal<string>();
    }

    public static string TaskName
    {
        get => taskName.Value;
        set => taskName.Value = value;
    }

    public static void WriteLine(string text)
    {
        Console.WriteLine($"{TaskName}: {text}");
    }
}
```

Локальные данные потока. Тип данных LocalDataStoreSlot

```
class Logger
{
    static readonly LocalDataStoreSlot slot;

    static Logger()
    {
        slot = Thread.AllocateDataSlot();
    }

    public static string TaskName
    {
        get => (string)Thread.GetData(slot);
        set => Thread.SetData(slot, value);
    }

    public static void WriteLine(string text)
    {
        Console.WriteLine($"{TaskName}: {text}");
    }
}
```

Домены приложений (AppDomain)

- В каждом процессе .NET можно создавать изолированные адресные пространства для работы с данными. Такие пространства называются доменами приложений (Application Domain).
- Домены приложений обеспечивают загрузку и выгрузку сборок и их данных.
- Передача данных (объектов) между доменами приложений осуществляется путем сериализации и десериализации.
- Начиная с .NET Core, поддерживается только один домен приложений для процесса.

Создание нового домена приложений.

Создание объекта в домене приложений.

Использование объекта из другого домена приложений

```
static void Main()
{
    AppDomain anotherDomain = AppDomain.CreateDomain("Домен-2");
    MyMarshalableByRefObject mbrt =
        (MyMarshalableByRefObject)anotherDomain.CreateInstanceAndUnwrap(
            Assembly.GetEntryAssembly().FullName,
            "Example.MyMarshalableByRefObject");
    Console.WriteLine("Тип данных: " + mbrt.GetType()); // CLR обманывает
    Console.WriteLine("Является посредником: " +
        RemotingServices.IsTransparentProxy(mbrt));
    mbrt.TestMethod();
    AppDomain.Unload(anotherDomain);
    try
    {
        mbrt.TestMethod();
    }
    catch (AppDomainUnloadedException)
    {
        Console.WriteLine("Ошибка: домен был выгружен.");
    }
}
```

Требования к объектам, создаваемым в других доменах приложений

```
public class MyMarshalableByRefObject : MarshalByRefObject
{
    public MyMarshalableByRefObject()
    {
        Console.WriteLine("Конструктор {0} запущен в домене {1}",
            this.GetType().ToString(), Thread.GetDomain().FriendlyName);
    }

    public void TestMethod()
    {
        Console.WriteLine("Метод TestMethod запущен в домене " +
            Thread.GetDomain().FriendlyName);
    }

    public MyMarshalableByValueObject CreateSerializableObject()
    {
        Console.WriteLine("Метод CreateSerializableObject запущен в домене " +
            Thread.GetDomain().FriendlyName);
        return new MyMarshalableByValueObject();
    }

    public MyNonMarshalableObject CreateNonSerializableObject()
    {
        Console.WriteLine("Метод CreateNonSerializableObject запущен в домене " +
            Thread.GetDomain().FriendlyName);
        MyNonMarshalableObject result = new MyNonMarshalableObject();
        Console.WriteLine("MyNonMarshalableObject объект создан");
        return result; // Ошибка при попытке вернуть объект в другой домен
    }
}
```

Передача объектов между доменами приложений

```
static void Main()
{
    AppDomain anotherDomain = AppDomain.CreateDomain("Домен-2");
    MyMarshalableByRefObject mbrt =
        (MyMarshalableByRefObject)anotherDomain.CreateInstanceAndUnwrap(
            Assembly.GetEntryAssembly().FullName,
            "Example.MyMarshalableByRefObject");
    MyMarshalableByValueObject mbvt = mbrt.CreateSerializableObject();
    Console.WriteLine("Является посредником: " +
        RemotingServices.IsTransparentProxy(mbvt));
    Console.WriteLine("Время создания объекта: " + mbvt.ToString());
    AppDomain.Unload(anotherDomain);
    Console.WriteLine("Время создания объекта: " + mbvt.ToString());
    Console.WriteLine("Успешное обращение к объекту после выгрузки домена.");
}
```

Требования к объектам, передаваемым между доменами приложений

```
[Serializable]
public class MyMarshalableByValueObject : Object
{
    private DateTime creationTime = DateTime.Now;

    public MyMarshalableByValueObject()
    {
        Console.WriteLine(
            "Конструктор {0} запущен в домене {1}, время: {2:D}",
            this.GetType().ToString(),
            Thread.GetDomain().FriendlyName,
            creationTime);
    }

    public override string ToString()
    {
        return creationTime.ToString("yyyy-MM-dd HH:mm:ss");
    }
}
```

Передача несериализуемых объектов между доменами приложений приводит к ошибке

```
static void Main()
{
    Console.WriteLine("Метод CreateNonSerializableObject запущен в домене "+
        Thread.GetDomain().FriendlyName);
    var result = new MyNonMarshalableObject();
    Console.WriteLine("MyNonMarshalableObject объект создан");
    return result; // Ошибка при попытке вернуть объект в другой домен
}

public class MyNonMarshalableObject : Object
{
    public MyNonMarshalableObject()
    {
        Console.WriteLine("Конструктор {0} запущен в потоке {1}",
            this.GetType().ToString(), Thread.GetDomain().FriendlyName);
    }
}
```

Асинхронные делегаты

- Класс делегата содержит методы BeginInvoke и EndInvoke.
- Метод BeginInvoke запускает делегат в пуле потоков и возвращает объект с интерфейсом IAsyncResult, который служит для ожидания окончания выполнения.
- Метод EndInvoke принимает объект с интерфейсом IAsyncResult и дожидается окончания выполнения делегата.
- Метод BeginInvoke имеет дополнительные параметры, позволяющие передать другой делегат, вызываемый по окончанию работы асинхронного делегата.
- Не поддерживается в .NET Core, .NET 5 и последующих версиях.

Асинхронные делегаты. Пример

```
class Program
{
    static void Main()
    {
        Action action = Do;
        IAsyncResult result = action.BeginInvoke(WorkCompleted, null);
        Console.WriteLine("Ожидание окончания выполнения делегата.");
        action.EndInvoke(result);
    }

    static void Do()
    {
        Console.WriteLine("Выполняется метод Do.");
        Thread.Sleep(1000);
    }

    static void WorkCompleted(object state)
    {
        Console.WriteLine("Работа выполнена.");
    }
}
```

Проблемы (асинхронных) делегатов, выполняемых с помощью пула потоков

- Нет стандартных средств возврата результата работы.
- Нет стандартных средств возврата информации об ошибке.
- Нет стандартных средств ожидания завершения.
- Нет стандартных средств досрочного прерывания работы.
- Нет стандартных средств ограничения степени параллелизма.
- Нет планирования и распределения процессорного времени.
- Нет возможности асинхронного (пошагового) выполнения в контексте одного (главного) потока.

Параллельные и асинхронные задачи на основе класса Task

- Свойство `Result` для хранения и возврата результата работы.
- Свойство `Exception` для хранения и возврата информации об ошибке.
- Метод `Wait` для ожидания завершения задачи.
- Параметр с типом `CancellationToken` для досрочного прерывания работы.
- Классы `Parallel` и `ParallelQuery` для моделирования систем массового обслуживания (СМО) и выполнения задач с заданной степенью параллелизма.
- Класс `TaskScheduler` для нестандартного планирования и распределения процессорного времени.
- `Async`-методы и оператор `await` в языке C# и компиляторе для асинхронного (пошагового) выполнения в контексте одного (например, главного) потока.
- Аналогами класса `Task` в языках Java и JavaScript являются классы `Future` и `Promise` соответственно.

Классы Task и Task<TResult>

```
public class Task : IAsyncResult, IDisposable
{
    public Task(Action action);
    public Task(Action<object> action, object state,
               CancellationToken cancellationToken, TaskCreationOptions creationOptions);
    public TaskCreationOptions CreationOptions { get; }
    public AggregateException Exception { get; }
    public bool IsCompleted { get; }
    public bool IsCanceled { get; }
    public object AsyncState { get; }
    public bool IsCompletedSuccessfully { get; }
    public int Id { get; }
    public bool IsFaulted { get; }
    public TaskStatus Status { get; }
    public void Start();
    public void Start(TaskScheduler scheduler);
    public void Wait();
    public bool Wait(int millisecondsTimeout, CancellationToken cancellationToken);
    public void RunSynchronously();
    public TaskAwaiter GetAwaiter();
    public ConfiguredTaskAwaitable ConfigureAwait(bool continueOnCapturedContext);
    public Task ContinueWith(Action<Task> continuationAction);
}

public class Task<TResult> : Task
{
    public TResult Result { get; }
}
```

Параллельные задачи, выполняемые в пуле потоков

```
static long Fibonachi(int count)
{
    long previous = 1; long result = 0;
    for (int i = 0; i < count; i++)
    {
        long s = previous + result;
        previous = result;
        result = s;
    }
    Console.WriteLine("ThreadId во время работы Fibonachi: {0}",
                      Thread.CurrentThread.ManagedThreadId);
    return result;
}

static Task<long> FibonachiTask(int count)
{
    Task<long> task = new Task<long>(() => Fibonachi(count));
    task.Start();
    return task;
}

static void Main(string[] args)
{
    Task<long> task = FibonachiTask(10);
    Console.WriteLine("Запустили задачу FibonachiTask");
    task.Wait();
    Console.WriteLine("Дождались задачу FibonachiTask, результат: {0}", result);
}
```

Асинхронные задачи, не требующие пула потоков

```
static async Task<long> FibonacciAsync(int count)
{
    Console.WriteLine("ThreadId во время работы FibonacciAsync: {0}",
        Thread.CurrentThread.ManagedThreadId);
    long result = await FibonacciTask(count);
    return result;
}

static void Main(string[] args)
{
    Console.WriteLine("Запускаем задачу FibonacciAsync");
    long result = FibonacciAsync(10).GetAwaiter().GetResult();
    Console.WriteLine("Дождались задачу FibonacciAsync, результат: {0}",
        result);
}
```

Асинхронное выполнение цепочки задач

```
static Task OutputPageText(string url)
{
    var httpClient = new HttpClient();
    return httpClient.GetStringAsync(url).ContinueWith(task =>
    {
        Console.WriteLine(task.Result);
    });
}

static async Task OutputPageTextAsync(string url)
{
    var httpClient = new HttpClient();
    string result = await httpClient.GetStringAsync(url);
    Console.WriteLine(result);
}
```

Компиляция асинхронных (async-) методов в конечный автомат

```
static async Task OutputPageTextAsync(string url)
{
    var httpClient = new HttpClient();
    string result = await httpClient.GetStringAsync(url);
    Console.WriteLine(result);
}
```



```
static Task OutputPageTextAsync_Compiled(string url)
{
    var _stateMachine = new _OutputPageTextAsync();
    _stateMachine._builder = AsyncTaskMethodBuilder.Create();
    _stateMachine.url = url;
    _stateMachine._state = -1;
    _stateMachine._builder.Start(ref _stateMachine);
    return _stateMachine._builder.Task;
}
```

Конечный автомат

```
public sealed class _OutputPageTextAsync : IAsyncStateMachine
{
    public string url;
    private HttpClient httpClient;
    private string result;

    // Состояние конечного автомата
    public int _state;
    // Хранит объект Task, возвращаемый из метода OutputPageText
    public AsyncTaskMethodBuilder _builder;
    // Объект, который служит для ожидания подзадач
    private TaskAwaiter<string> _awaiter;

    public void MoveNext()
    {
        ...
    }

    public void SetStateMachine(IAsyncStateMachine stateMachine)
    {
    }
}
```

Конечный автомат. Метод MoveNext

```
public void MoveNext()
{
    try
    {
        TaskAwaiter<string> awaiter;
        if (_state != 0)
        {
            // Код, запускаемый до await:
            httpClient = new HttpClient();
            // Код, соответствующий оператору await:
            awaiter = httpClient.GetStringAsync(url).GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                _state = 0;
                _awaiter = awaiter;
                _OutputPageText stateMachine = this;
                // В очередь к задаче ставится вызов этого же метода (MoveNext):
                _builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
                return;
            }
        }
        else
        {
            awaiter = _awaiter;
            _awaiter = new TaskAwaiter<string>();
            _state = -1;
        }
        result = awaiter.GetResult();
        // Код, запускаемый после оператора await:
        Console.WriteLine(result);
    }
    catch (Exception exception)
    {
        _state = -2;
        httpClient = null;
        result = null;
        // Завершение задачи с ошибкой
        _builder.SetException(exception);
        return;
    }
    _state = -2;
    httpClient = null;
    result = null;
    // Успешное завершение задачи:
    _builder.SetResult();
}
```

Итераторы: интерфейсы IEnumerator и IEnumerable, оператор foreach

- Итератор – приём программирования для единообразной обработки элементов коллекций независимо от их разновидности и реализации (массив, связный список, словарь, множество).
- Оператор foreach транслируется компилятором языка C# в цикл while, использующий итератор.

```
public interface IEnumerator<out T> : IDisposable
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

```
public interface IEnumerable<out T>
{
    IEnumerator<T> GetGetEnumerator();
}
```

Оператор foreach для массива

```
static void Main()
{
    int[] numbers = new int[5] { 0, 1, 2, 3, 4 };

    foreach (int a in numbers)
    {
        Console.WriteLine(a);
    }

    IEnumerator it = numbers.GetEnumerator();
    while (it.MoveNext())
    {
        int a = (int)it.Current;
        Console.WriteLine(a);
    }
}
```

Оператор foreach для списка

```
static void Main()
{
    var numberList = new List<int> { 0, 1, 2, 3, 4 };

    foreach (int a in numberList)
    {
        Console.WriteLine(a);
    }

    using (IEnumerator<int> it = numberList.GetEnumerator())
    {
        while (it.MoveNext())
        {
            int a = it.Current;
            Console.WriteLine(a);
        }
    }
}
```

Пример создания итератора для списка строк

```
public class StringList : List<string>
{
    public IEnumerable<string> GetTop(int topCount)
    {
        return new StringListTopCountEnumerable(this, topCount);
    }
}

public struct StringListTopCountEnumerable : IEnumerable<string>
{
    private StringList stringList;
    private int topCount;

    public StringListTopCountEnumerable(StringList stringList, int topCount)
    {
        this.stringList = stringList;
        this.topCount = topCount;
    }

    public IEnumerator<string> GetEnumerator()
    {
        return new StringListTopCountEnumerator(stringList, topCount);
    }

    IEnumerator IEnumerable.GetEnumerator() { return GetEnumerator(); }
}
```

Пример создания итератора для списка строк

```
public struct StringListTopCountEnumerator : IEnumerator<string>
{
    private StringList stringList;
    private int currentIndex;
    private int topCount;

    public StringListTopCountEnumerator(StringList stringList, int topCount)
    {
        this.stringList = stringList;
        this.topCount = topCount;
        currentIndex = -1;
    }

    public string Current => stringList[currentIndex];
    object IEnumerator.Current => Current;

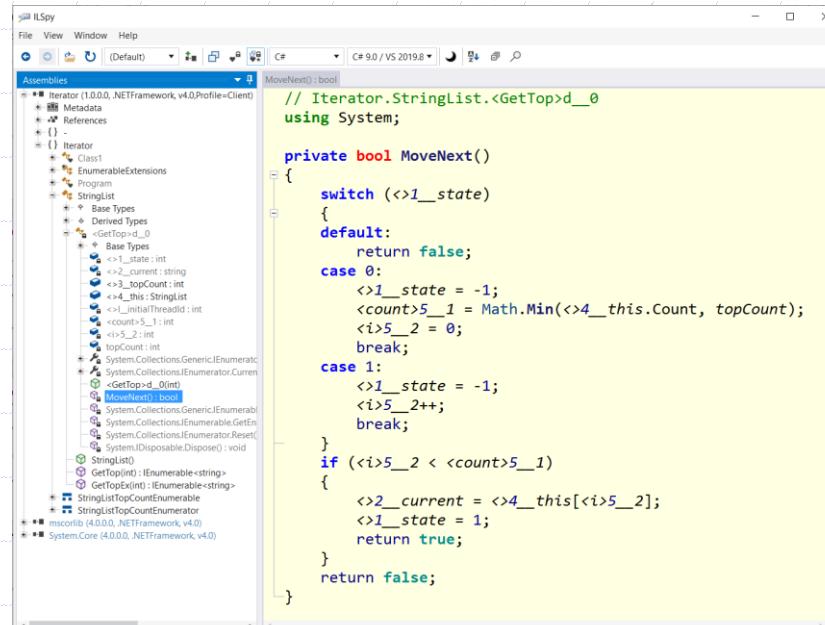
    public bool MoveNext()
    {
        currentIndex++;
        return currentIndex < stringList.Count && currentIndex < topCount;
    }

    public void Reset() { currentIndex = -1; }

    public void Dispose() { }
}
```

Оператор yield return

```
public class StringList : List<string>
{
    public IEnumerable<string> GetTop(int topCount)
    {
        int count = Math.Min(Count, topCount);
        for (int i = 0; i < count; i++)
        {
            yield return this[i];
        }
    }
}
```



Асинхронные итераторы. Интерфейсы IAsyncEnumerable и IAsyncEnumerator

- Асинхронные итераторы – это асинхронные методы на основе оператора yield return.
- Асинхронное итерирование с ожиданием выполняется с помощью оператора await foreach.

```
public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}

public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}
```

Асинхронные итераторы. Пример

```
static async IAsyncEnumerable<long> GetFibonachiList(int count)
{
    for (int i = 0; i < count; i++)
        yield return await FibonachiAsync(i);
}

static async Task WriteFibonachiList(int count)
{
    var stream = GetFibonachiList(count);
    await foreach (long x in stream)
    {
        Console.WriteLine("Fibonachi: " + x);
    }
}

static async Task WriteFibonachiList(int count)
{
    var stream = GetFibonachiList(count);
    IAsyncEnumerator<long> enumerator = stream.GetAsyncEnumerator();
    while (await enumerator.MoveNextAsync())
    {
        long x = enumerator.Current;
        Console.WriteLine("Fibonachi: " + x);
    }
}
```

Коллекции со встроенными средствами синхронизации доступа

- Находятся в пространстве имён `System.Collections.Concurrent` и имеют имена, начинающиеся со слова `Concurrent`:
 - `ConcurrentDictionary` – словарь;
 - `ConcurrentQueue` – очередь;
 - `ConcurrentStack` – стек;
 - `ConcurrentBag` – неупорядоченная коллекция с поддержкой дубликатов.
- Как правило работают более эффективно, чем коллекции, над которыми применяется оператор `lock`.

Особенности работы с коллекцией ConcurrentDictionary

```
static void Main()
{
    var collection = new ConcurrentDictionary<int, string>();

    string value = collection.GetOrAdd(5, "Невод");
    Console.WriteLine(value);

    string newValue = "Незабудка";
    collection.AddOrUpdate(5, newValue,
        (int key, string oldValue) =>
    {
        Console.WriteLine($"Заменяем {oldValue} на {newValue}.");
        return newValue;
    });

    if (collection.TryGetValue(5, out value))
        Console.WriteLine(value);
}
```

Использование класса ParallelQuery

```
static void Main()
{
    int[] numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    List<int> result = numbers
        .AsParallel()
        .WithDegreeOfParallelism(4)
        .Where(n => n % 2 == 0)
        .Select(Square).ToList();

    result.ForEach(x => Console.WriteLine(x));
}

static int Square(int n)
{
    Console.WriteLine("Обработка {0} потоком {1}",
        n, Thread.CurrentThread.ManagedThreadId);
    return n * n;
}
```

Контекст синхронизации

- В программах с пользовательским интерфейсом только один поток создает элементы пользовательского интерфейса и выполняет цикл обработки сообщений окна.
- Для выполнения длительных действий без остановки обработки сообщений (без блокировки пользовательского интерфейса) используются дополнительные потоки.
- Обращаться к элементам пользовательского интерфейса можно только из потока, который их создал (главного потока). Поэтому отображение на экране результатов длительных действий необходимо выполнять только в контексте главного потока.
- Контекст синхронизации позволяет выполнять действия в контексте главного потока:
 - С помощью метода `Send` в очередь сообщений окна ставится специальное сообщение со ссылкой на заданный делегат.
 - При обработке такого сообщения главным потоком вызывается делегат, содержащийся в параметрах сообщения.

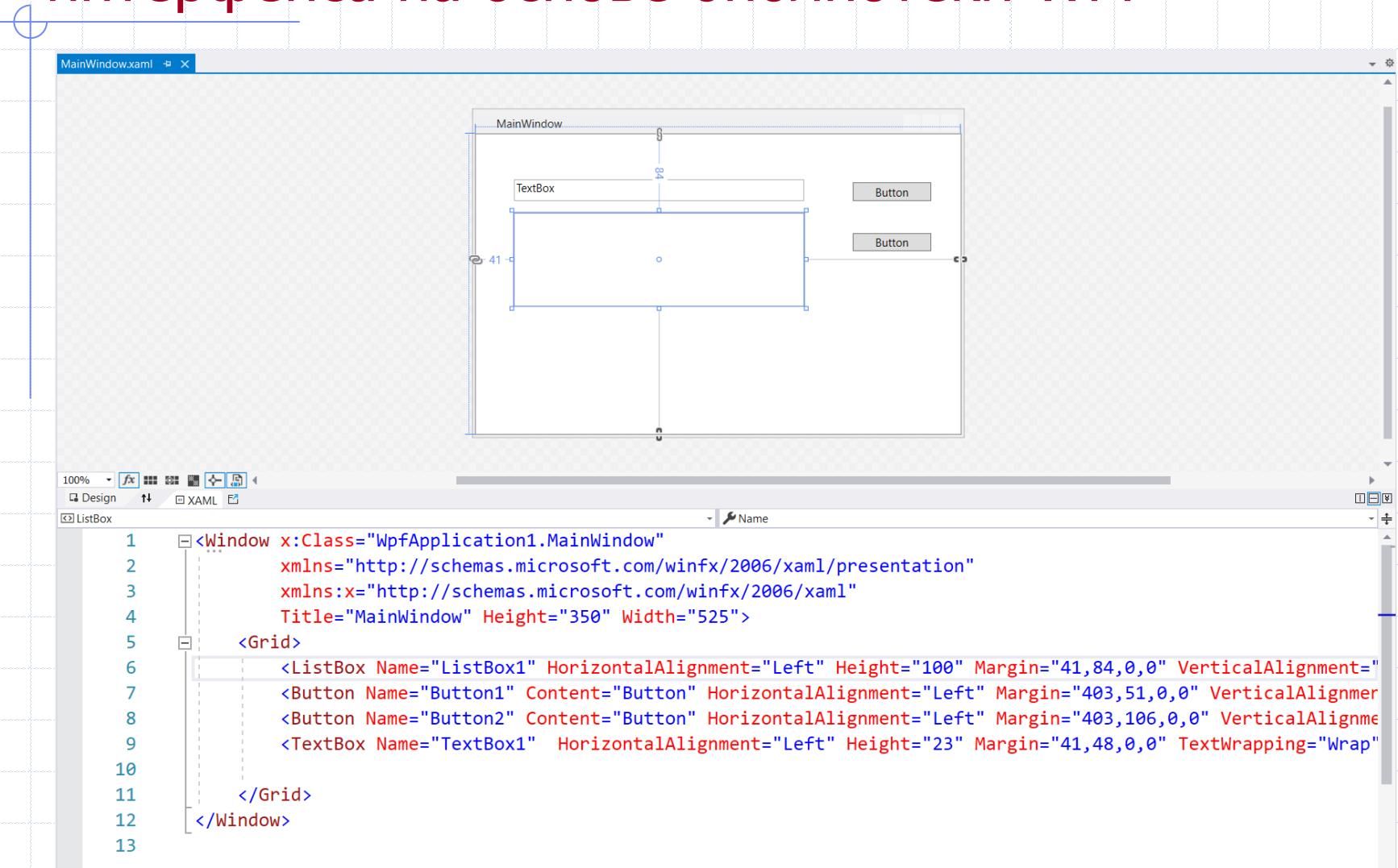
Контекст синхронизации в пользовательском интерфейсе на основе библиотеки WPF

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void AddItemViaSynchronizationContext(object state)
    {
        var context = (SynchronizationContext)state;
        context.Send((x) =>
        {
            ListBox1.Items.Add("text");
        }, null);
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        ThreadPool.QueueUserWorkItem(AddItemViaSynchronizationContext,
            SynchronizationContext.Current);
    }
}
```

Контекст синхронизации в пользовательском интерфейсе на основе библиотеки WPF



Взаимодействие с неуправляемым кодом

```
[StructLayout(LayoutKind.Sequential)]
public class SystemTime
{
    public UInt16 wYear;
    public UInt16 wMonth;
    public UInt16 wDayOfWeek;
    public UInt16 wDay;
    public UInt16 wHour;
    public UInt16 wMinute;
    public UInt16 wSecond;
    public UInt16 wMilliseconds;
}

[DllImport("kernel32", EntryPoint = "GetSystemTime",
    CallingConvention = CallingConvention.Winapi,
    CharSet = CharSet.Unicode)]
public extern static void Get(SystemTime result);
}

class Program
{
    static void Main()
    {
        var t = new SystemTime();
        SystemTime.Get(t);
        Console.WriteLine(
            "{0}-{1}-{2} {3}:{4}:{5}",
            t.wDay, t.wMonth, t.wYear,
            t.wHour, t.wMinute, t.wSecond);
    }
}
```

Наложение разнотипных полей друг на друга

```
[StructLayout(LayoutKind.Explicit)]
public struct TwoBytes
{
    [FieldOffset(0)]
    private byte FirstByte;
    [FieldOffset(1)]
    private byte SecondByte;
    [FieldOffset(0)]
    private ushort Word;

    public static bool IsLittleEndian()
    {
        var twoBytes = new TwoBytes { Word = 1 };
        return twoBytes.FirstByte == 1;
    }
}
```

Тип данных dynamic

- Применение типизации для объектов с типом `dynamic` выполняется динамически во время работы программы.
- Обращение к свойству или методу транслируется компилятором в вызов диспетчера с передачей имени и другой информации, необходимой для поиска свойства или метода.
- Допускается создание типов данных, реализующих собственную логику диспетчеризации доступа к методам и свойствам. Такие типы данных должны быть производными от `DynamicObject`.

Использование dynamic-объектов для работы с данными в формате JSON

```
static void Main()
{
    string json =
        "{\"Country\": \"Беларусь\", \"City\": \"Минск\"}";
    dynamic obj =
        Newtonsoft.Json.JsonConvert.DeserializeObject(json);
    Console.WriteLine(obj.Country);
    Console.WriteLine(obj.City);
}
```

Реализация собственного dynamic-объекта

```
public class DynamicWriter : DynamicObject
{
    private void DoWriteLine(object arg)
    {
        Console.WriteLine(arg);
    }

    public void WriteLine(object arg)
    {
        Console.WriteLine("Вызван WriteLine");
    }

    public override bool TryInvokeMember(InvokeMemberBinder binder,
                                         object[] args, out object result)
    {
        bool memberFound = false;
        result = null;
        if (binder.Name == "WriteLine")
        {
            Console.WriteLine("Вызван TryInvokeMember");
            DoWriteLine(args[0]);
            memberFound = true;
        }
        return memberFound;
    }
}
```

Динамическая генерация кода на основе деревьев выражений

- Дерево выражения – это дерево, кодирующее синтаксис выражения на высокоуровневом языке программирования, который схож с языком C#.
- Для кодирования используется иерархия классов, производных от класса Expression в пространстве имен System.Linq.Expressions.
- Корневым узлом дерева выражения является так называемое «лямбда-выражение», которое можно скомпилировать в подпрограмму с параметрами и результатом.
- Компилятор C# поддерживает конструирование деревьев выражений из фрагментов программного кода.

Динамическая генерация кода на основе деревьев выражений. Пример

```
public static Func<int, bool> CreateFunctionFromCSharp()
{
    Expression<Func<int, bool>> lambda = num => num < 5;
    Func<int, bool> function = lambda.Compile();
    return function;
}

public static Func<int, bool> CreateFunctionUsingExpressionsAPI()
{
    ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
    ConstantExpression five = Expression.Constant(5, typeof(int));
    BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
    Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(
        numLessThanFive, numParam);
    Func<int, bool> function = lambda.Compile();
    return function;
}
```

Динамическая генерация кода на основе деревьев выражений. Пример вызова метода

```
public static Action<int> CreateProcedureFromCSharp()
{
    Expression<Action<int>> lambda = (arg) => Console.WriteLine(arg);
    Action<int> procedure = lambda.Compile();
    return procedure;
}

public static Action<int> CreateProcedureUsingExpressionsAPI()
{
    ParameterExpression param = Expression.Parameter(
        typeof(int), "arg");
    MethodCallExpression methodCall = Expression.Call(
        typeof(Console).GetMethod("WriteLine", new Type[] { typeof(int) }), param);
    Expression<Action<int>> lambda = Expression.Lambda<Action<int>>(
        methodCall, param);
    Action<int> procedure = lambda.Compile();
    return procedure;
}
```

Динамическая генерация кода на основе деревьев выражений. Пример блока кода

```
static int Factorial(int value)
{
    int result = 1;
    while (value > 1)
        result *= value--;
    return result;
}

public static Func<int, int> CreateFactorialProcedure()
{
    ParameterExpression value = Expression.Parameter(typeof(int), "value");
    ParameterExpression result = Expression.Parameter(typeof(int), "result");
    LabelTarget label = Expression.Label(typeof(int));
    BlockExpression block = Expression.Block(new[] { result },
        Expression.Assign(result, Expression.Constant(1)),
        Expression.Loop(
            Expression.IfThenElse(
                Expression.GreaterThan(value, Expression.Constant(1)),
                Expression.MultiplyAssign(result,
                    Expression.PostDecrementAssign(value)),
                Expression.Break(label, result)
            ),
            label)
    );
    var lambda = Expression.Lambda<Func<int, int>>(block, value);
    Func<int, int> procedure = lambda.Compile();
    return procedure;
}
```



Конец

Д.А. Сурков:

dmitry.surkov@gmail.com

И.В. Шимко:

bsuir@ishimko.me