МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по Microsoft .NET и языку программирования C#

Учебно-методическое пособие

Часть 4. Домены, приложения, сборки и потоки

2018 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по Microsoft .NET и языку программирования C#. Учебно-методическое пособие. Часть 4. Домены, приложения, сборки и потоки. Озерск: ОТИ НИЯУ МИФИ, 2018. — 21 с.

В пособии предлагаются практические работы по изучению платформы Microsoft .NET и языка программирования C#.

В этой части рассматриваются домены, приложения, сборки и потоки.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1. Синяков В. Е., начальник УИТ ФГУП «ПО «Маяк».
2. Зубаиров А. Ф., ст. преподаватель кафедры ПМ ОТИ НИЯУ МИФИ.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

- домены, приложения и сборки .NET;
- потоки и синхронизация.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

Примеры программ данного сборника работ заимствованы из книги «Троэлсен Э. C# и платформа .NET. СПб.: Питер, 2005. — 796 с.: ил.».

1. Работа ООП-401. Сборки и приложения DOT NET

Цели:
- управления сборками и приложениями DOT NET.
Задачи:
- построение библиотеки;
- управление сборками;
- управление приложениями.

Опорные документы:

1.1. Библиотека классов

Создадим проект приложения C# типа Class Library (dll).
Название проекта — carlib.
В пространстве имен carlib описываем перечисление и три класса.
Перечисление:

```
public enum EngineState {
    engineAlive,
    engineDead
}
```

Первый класс абстрактный:

```
public abstract class Car {
    public string name = "noname";
    public short maxSpeed = 0;
    public short speed = 0;
    public EngineState state = EngineState.engineAlive;
    // конструктор по умолчанию
    public Car() { }
    // конструктор
    public Car(string name, short maxspeed, short speed) {
        this.name = name;
        this.maxSpeed = maxspeed;
        this.speed = speed;
    }
    // абстрактный
    public abstract void TurboBoost();
}
```

Новый класс наследует Car:

```
public class SportsCar : Car {
    public SportsCar() { }
    public SportsCar(string name, short max, short speed)
        : base(name, max, speed) { }
    public override void TurboBoost() {
        Console.WriteLine("Faster is better.");
    }
}
```

Еще один класс наследует Car:

```
public class MiniVan : Car {
    public MiniVan() { }
    public MiniVan(string name, short max, short speed)
        : base(name, max, speed) { }
    public override void TurboBoost() {
        Console.WriteLine("The car is dead.");
    }
}
```

Обратим внимание на public в начале всех структур.

Компилируем dll.

1.2. Клиентское приложение

Создаем новое консольное приложение cars другой копией студии (не закрывая проект carlib). Компилируем его.

При помощи диалога Project — Add Reference... — Browse найдем и выберем carlib.dll.

Используем пространство имен carlib:

```
namespace cars {
    using carlib;
    . . .
}
```

Далее используем классы библиотеки, например, так:

```
    static void Main(string[] args) {
        SportsCar viper = new SportsCar("Viper", 240, 40);
        viper.TurboBoost();
        MiniVan mv = new MiniVan();
        mv.TurboBoost();
    }
```

Запускаем программу, убеждаемся, что кары создаются.

1.3. Сборка общего доступа

Любая сборка является частной (private) или общей (shared) — третьего не дано. Частные сборки — это наборы типов, которые входят только в те приложения, в которых они используются. Частные сборки находятся в основном каталоге приложения-владельца или в его подкаталогах.

Основным каталогом приложения-владельца является каталог сборки-владельца. Убедимся в том, что сборка carlib.dll действительно находится в каталоге владельца. Размещение копии частной сборки происходит автоматически.

Частная сборка идентифицируется по дружественному имени и версии. Для частной сборки .NET не использует никакой политики в отношении версий.

Откроем FAR и найдем исполняемый файл сборки cars.exe. Каталог, в котором файл найдется — каталог приложения cars.

Создадим F7 подкаталог libs и переместим в него сборку carlib.dll.

Запустим приложение cars.exe (из FAR, нажав Enter).

Не работает, выскакивает исключение.

Работаем в FAR. Каталог приложения (там, где cars.exe). Shift+F4, название файла cars.exe.config, Enter, вводим следующий текст xml:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="libs"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

Запустим приложение cars.exe.

Работает. Таким образом, файл конфигурации может быть использован для указания пути к используемым сборкам.

Файл конфигурации нужно переименовать в cars.exe.config.1.

1.4. Сборка общего доступа

Сборки для общего доступа (shared) — это набор типов и необязательных ресурсов, как и для частной сборки. Главное отличие от частных сборок заключается в том, что сборки общего доступа предназначены для использования любыми приложениями. Сборки общего доступа устанавливаются в специальный каталог, называемый «глобальный кэш сборок» GAC (Global Assembly Cache).

Кроме того, сборки общего доступа отличаются дополнительной информацией о версии. Эта дополнительная информация называется строгим именем (strong name). Информацию о версии сборки среда .NET не игнорирует, а активно использует.

При создании сборки общего доступа требуется создать строгое имя.

Создание строгого имени сборки основано на криптографии открытого ключа. При этом необходимо создать пару открытый/закрытый ключ.

Компилятор поместит маркер открытого ключа в манифест сборки, в тег .publickeytoken. Закрытый ключ используется для создания цифровой подписи, которая помещается в сборку. Закрытый ключ хранится в том модуле сборки, в котором находится манифест.

При создании сборки общего доступа компилятор помещает открытый ключ в манифест. Во время выполнения .NET проверяет соответствие маркера открытого ключа сборки, запрашиваемой клиентом, и маркера открытого ключа в самой сборке общего доступа.

Такая проверка гарантирует, что клиент получит именно ту сборку общего доступа, которая ему нужна (процесс описан в самом общем виде).

Создание пары открытый/закрытый ключ производится при помощи утилиты sn.exe. Сейчас нужно найти на компьютере файлы sn.exe и gacutil.exe, и скопировать их в каталог библиотеки (там, где carlib.dll) для удобства выполнения последующих действий.

Выполним следующую команду с клавиатуры:

```
sn -k keys.snk
```

Проект carlib.

Выберем в меню Project — carlib Properties, выберем вкладку Signing.

Включим флажок Sign the assembly.

В поле Choose a strong name key file выберем Browse, найдем и выберем файл keys.snk.

Скомпилируем и сохраним проект.

Откроем утилиту ILdasm.exe, откроем сборку carlib.

Убедимся, что в ней появился тег .publickey, открытый ключ.

Теперь установим новую сборку в GAC, для чего введем с клавиатуры команду:

```
gacutil -i carlib.dll
```

Если сборка успешно установлена, в консоли выводится сообщение «Assembly successfully added to the cache».

Перейдем в проект cars.

Перенесем оператор using carlib:

```
using System;
using carlib;
namespace cars {
    class Program {
        static void Main(string[] args) {
            SportsCar viper = new SportsCar("Viper", 240, 40);
            viper.TurboBoost();
            MiniVan mv = new MiniVan();
            mv.TurboBoost();
        }
    }
}
```

Скомпилируем и запустим проект...

Откроем браузер объектов Ctrl+Alt+J, выберем в нем carlib и увидим примерно следующую информацию о сборке:

```
Assembly carlib
C:\WINDOWS\assembly\GAC_MSIL\carlib\1.0.0.0__d249c4d98f522120\carlib.dll
```

1.5. Политика версий

Перейдем в проект carlib. Добавим новый класс VWMiniVan:

```
class VWMiniVan : MiniVan {
    public VWMiniVan() { }
    public VWMiniVan(string name, short max, short curr)
        : base(name, max, curr) {
    }
    public void IsVersion() {
        Console.WriteLine("Version is 1.0.0.0.");
    }
}
```

Скомпилируем и установим сборку в GAC.
Перейдем в проект cars.
Добавим вызов нового класса в Main:

```
static void Main(string[] args) {
    SportsCar viper = new SportsCar("Viper", 240, 40);
    viper.TurboBoost();
    MiniVan mv = new MiniVan();
    mv.TurboBoost();
    VWMiniVan vmv = new VWMiniVan();
    vmv.IsVersion();
}
```

Запустим программу, убедимся, что выводится версия 1.0.0.0.
Перейдем в проект carlib.
Откроем файл AssemblyInfo.cs.
Исправим версию сборки на 2.0.0.0:

```
[assembly: AssemblyVersion("2.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Исправим номер версии в методе IsVersion:

```
public void IsVersion() {
    Console.WriteLine("Version is 2.0.0.0.");
}
```

Скомпилируем и еще раз установим сборку в GAC.
Откроем проект cars.
Запустим проект и убедимся, что работает версия 2.0.0.0.
Откроем основной каталог cars.
Сделаем копию cars.exe.config файла cars.exe.config.1.
Откроем файл cars.exe.config.
Изменим его:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="carlib"
            publicKeyToken="d249c4d98f522120"
            culture="" />
        <bindingRedirect oldVersion="2.0.0.0"
                         newVersion="1.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Здесь d249c4d98f522120 — число, отображаемое в браузере объектов, когда выбирается библиотека carlib.dll.
Запустим файл cars.exe.
Теперь работает версия 1.0.0.0.
Закроем все проекты.

1.6. Управления приложениями и рефлексия типов

Создадим новое консольное приложение CarDomain.

Уточним using:

```
using System;
using System.Reflection;
using System.Security.Permissions;
```

Добавим атрибут перед функцией Main:

```
[PermissionSetAttribute(SecurityAction.Demand, Name = "FullTrust")]
public static void Main(string[] args) {
}
```

Для работы со сборками используется рефлексия и класс Assembly.

Для работы с доменами используется класс AppDomain.

Сначала мы выведем список всех загруженных сборок.

Метод Main:

```
AppDomain ad = AppDomain.CurrentDomain;
Assembly[] loaded = ad.GetAssemblies();
foreach (Assembly s in loaded) {
    Console.WriteLine(s.FullName);
}
```

Запустим программу, увидим в консоли примерно следующее:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c56193
carDomain, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
```

Теперь в начале функции Main добавим код для загрузки сборки, которую укажем параметром командной строки:

```
public static void Main(string[] args) {
    Assembly a = null;
    try {
        a = Assembly.Load(args[0]);
    } catch (Exception e) {
        Console.WriteLine(e.Message);
    }

    AppDomain ad = AppDomain.CurrentDomain;
    Assembly[] loaded = ad.GetAssemblies();
    foreach (Assembly s in loaded) {
        Console.WriteLine(s.FullName);
    }
}
```

Скомпилируем проект.

Перейдем в основной каталог проекта. Скопируем в него приложение cars.exe. Запустим программу CarDomain, указывая название cars:

```
путь>CarDomain.exe cars
```

Запустим программу, увидим в консоли примерно следующее:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c56193
carDomain, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
cars, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
```

При помощи класса Type мы можем получить информацию о типах сборки. После цикла foreach:

```
public static void Main(string[] args) {
    . . .
    Type[] mytypes = a.GetTypes();
    BindingFlags flags = (
        BindingFlags.NonPublic |
        BindingFlags.Public |
        BindingFlags.Static |
        BindingFlags.Instance |
        BindingFlags.DeclaredOnly);
    foreach (Type t in mytypes) {
        MethodInfo[] mi = t.GetMethods(flags);
        foreach (MethodInfo m in mi) {
            Console.WriteLine(m);
        }
    }
}
```

Компилируем проект и запускаем в командной строке, указывая параметр. Рекомендуется скопировать carlib.dll в основной каталог проекта и использовать имя dll в качестве параметра. Можно установить параметры командной строки в свойствах проекта и запускать его в среде разработки.

Вывод в консоли примерно следующий:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934
carDomain, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
carlib, Version=1.0.0.0, Culture=neutral,
publicKeyToken=d249c4d98f522120
System.String get_Name()
Int16 get_MaxSpeed()
Int16 get_Speed()
Void set_Speed(Int16)
CarLib.EngineState get_EngineState()
Void TurboBoost()
Void TurboBoost()
Void TurboBoost()
Void IsVersion()
```

1.6.1. Позднее связывание метода сборки

Используя информацию о типах, можно применить позднее связывание. Сначала получим тип:

```
// позднее связывание с методом
Console.WriteLine();
Type mv = a.GetType("carlib.MiniVan");
```

Далее создаем объект класса MiniVan, используя класс Activator, получаем метод (информацию о методе) TurboBoost, и, используя позднее связывание, вызываем этот метод:

```
Object obj = Activator.CreateInstance(mv);
MethodInfo mvmi = mv.GetMethod("TurboBoost");
mvmi.Invoke(obj, null);
```

Запускаем программу, убеждаемся, что вызывается метод TurboBoost.

1.6.2. Динамические сборки

Сборки бывают статическими и динамическими.

Статические сборки имеют вид файлов.

Динамические сборки создаются во время работы других сборок.

Ввиду сложности данного процесса здесь он не описывается (требуется знание IL). Последнее, что мы сделаем — создадим новый домен и загрузим в него сборку.

Создаем новый домен (метод Main, продолжение):

```
// динамическая сборка
Console.WriteLine();
AppDomain dm = AppDomain.CreateDomain("CARS");
```

Загружаем в него сборку cars, выполняем ее, выгружаем домен:

```
dm.ExecuteAssembly("cars.exe");
AppDomain.Unload(dm);
```

Запускаем программу, убеждаемся, что выполняется сборка целиком.

1.7. Контрольные вопросы и упражнения

1. Какая сборка называется частной, какая сборкой общего доступа?
2. Где размещается частная сборка?
3. Где размещается сборка общего доступа?
4. Что называется строгим именем?
6. Какова политика версий в отношении сборок общего доступа?
7. Для чего нужен файл конфигурации приложения?
8. Что называется доменом приложения?
9. В чем заключается позднее связывание с методом?
10. Что называется динамической сборкой?

2. Работа ООП-402. Управление потоками

Цели:
- управления потоками в DOT NET.
Задачи:
- создание потоков;
- управление потоками;
- синхронизация ключевым словом lock;
- синхронизация монитором;
- синхронизация атрибутом;
- wait-методы и события.
Опорные документы:

2.1. Создание потока

Здесь мы продемонстрируем управление потоком и некоторые важные методы класса Thread.

Создадим новое консольное приложение C#.

Название проекта SharpThreads.

Уточним инструкции using:

```
using System;
using System.Threading;
namespace SharpThreads {
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

Потоком является метод какого-либо класса, желательно без параметров. Создаем класс потока Worker:

```
namespace SharpThreads {
    internal class Worker {
        // признак работы потока
        private volatile bool running = false;
        // метод - функция потока
        public void process() {
        }
        // метод, сигнализирующий о завершении
        public void stop_running() {
            running = false;
        }
    }
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

Метод process:

```
public void process() {
    Console.WriteLine("now enter...");
    running = true;
    while (running) {
        Console.WriteLine("...processing...");
        Thread.Sleep(1);
    }
    Console.WriteLine("now leave...");
}
```

Переходим к методу Main.

Создаем объект и поток, стартуем поток:

```
static void Main(string[] args) {
    Console.WriteLine("Main starts.");
    // класс потока
    Worker w = new Worker();
    // создаем поток
    Thread t = new Thread(new ThreadStart(w.process));
    // стартуем поток
    t.Start();
}
```

Здесь для создания потока использован делегат ThreadStart. Другой способ создать поток — не использовать делегат (просто new Thread()).

Следующая строчка кода ожидает, когда поток начнет выполнение:

```
// ждем активизации
while (!t.IsAlive) ;
```

Далее приостанавливаем первичный поток на некоторое время:

```
// даем время поработать, пока основной поток спит
Thread.Sleep(100);
```

Теперь останавливаем вторичный поток:

```
// запрашиваем завершение потока
w.stop_running();
```

и ждем его завершения, приостанавливая выполнение первичного потока при помощи метода Join:

```
// ждем завершения потока, приостанавливая основной
t.Join();
Console.WriteLine("Main ends.");
```

Возможно, потребуется подобрать подходящие времена задержки потоков для того, чтобы проследить выполнение первичного и вторичного потоков.

2.2. Поток с параметрами

Создадим новое консольное приложение C#.

Название проекта SharpThreads2.

Используем следующие инструкции using:

```
using System;
using System.Threading;
```

Здесь мы исследуем использование потока с параметрами.

Существует делегат ParameterizedThreadStart, однако в MSDN2005 отмечается его потенциальная опасность в отношении типов. (Может быть, в новых версиях .NET Framework ситуация улучшена.) Поэтому, следуя той же MSDN, создается оберточный класс, хранящий параметры потока.

Описываем оберточный класс:

```
internal class ThreadWithState {
    // информация, необходимая потоку
    private string sos;
    private int sin;
}
```

Далее описываем конструктор:

```
    // конструктор получает необходимую информацию
    public ThreadWithState(string s, int n) {
        sos = s;
        sin = n;
    }
```

Описываем метод, выполняющий поток:

```
    // функция потока
    public void ThreadProc() {
        Console.WriteLine(sos, sin);
    }
```

Переходим к методу Main:

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

Далее создаем объект класса потока и передаем параметры:

```
        // конструктор получает параметры
        ThreadWithState w = new ThreadWithState("Number {0}", 33);
```

Создаем поток, запускаем его и ждем начала работы:

```
        Thread t = new Thread(new ThreadStart(w.ThreadProc));
        t.Start();
        while (!t.IsAlive) ;
```

Наконец, ждем воссоединения потоков и завершаем:

```
        Console.WriteLine("Main works and waits.");
        t.Join();
        Console.WriteLine("Main ends.");
```

Запускаем и смотрим, что происходит.

2.3. Параллельная работа потоков

Создадим новое консольное приложение C#.
Название проекта SharpThreads3.
Используем следующие инструкции using:

```
using System;
using System.Threading;
```

Описываем класс Worker, метод DoWork которого будет исполняемым потоком:

```
internal class Worker {
    public void DoWork() {
        // Current thread info
        Console.WriteLine("ID of Worker is: {0}",
            Thread.CurrentThread.GetHashCode());
        Console.WriteLine("Worker says: ");
        for (int i = 0; i < 10; i++) {
            Console.Write(i + ", ");
        }
        Console.WriteLine();

    }
}
```

Метод Main создает объект w, поток t и запускает его:

```
        // Current thread info
        Console.WriteLine("ID of Main is: {0}",
            Thread.CurrentThread.GetHashCode());
        Worker w = new Worker();
        Thread t = new Thread(new ThreadStart(w.DoWork));
        t.Start();
        t.Join();
```

Запускаем приложение, наблюдаем работу потока.

Сложно убедиться в параллельной работе первичного потока метода Main и вторичного потока метода DoWork. Изменим класс Worker так, чтобы он выводил на консоль не 10 чисел, а, например, 1000:

```
    public void DoWork() {
        // Current thread info
        Console.WriteLine("ID of Worker is: {0}",
            Thread.CurrentThread.GetHashCode());
        Console.WriteLine("Worker says: ");
        for (int i = 0; i < 1000; i++) {
            Console.Write(i + ", ");
        }
        Console.WriteLine();

    }
```

Соответственно, добавим работу для первичного потока после того, как стартует вторичный поток:

```
        t.Start();
        // Main thread works
        for (int i = 0; i < 500; i++) {
            Console.WriteLine("main " + i);
        }
        t.Join();
```

Запустим проект и проверяем, как происходит параллельная работа потоков. Вероятно, потребуется подобрать числа итераций первичного и (или) вторичного потоков, чтобы хоть что-то разглядеть и понять, что, когда и каким потоком выводится.

2.4. Синхронизация потоков

При работе с потоками, использующими разделяемые данные, важной задачей является обеспечение целостности этих данных. Один поток может быть остановлен во время работы с элементом данных, а другой поток изменит элемент данных так, что предыдущий поток продолжит работу с измененным элементом.

Прежде хотелось бы показать, что несколько потоков конкурируют друг с другом. Для этой цели создадим множество потоков, которые используют одни и те же данные, точнее, — одно и то же устройство вывода (консоль).

Создадим новое консольное приложение C#.

Название проекта SharpThreadsSyn.

Новый проект имеет ту же структуру, что и предыдущий.

Класс Worker:

```
internal class Worker {
    private string name = "noname";
    public Worker(string name) {
        this.name = name;
    }
    public void DoWork() {
        for (int i = 0; i < 5; i++) {
            Console.Write("Worker ");
            Thread.Sleep(0);
            Console.Write(name);
            Console.Write("-");
            Console.Write(i + " ");
            Thread.Sleep(0);
            Console.WriteLine();
        }
    }
}
```

В Main создается объект класса Worker и запускаются три потока:

```
    static void Main(string[] args) {
        Worker w1 = new Worker("A");
        Thread wA = new Thread(new ThreadStart(w1.DoWork));
        Worker w2 = new Worker("B");
        Thread wB = new Thread(new ThreadStart(w2.DoWork));
        Worker w3 = new Worker("C");
        Thread wC = new Thread(new ThreadStart(w3.DoWork));
        wA.Start();
        wB.Start();
        wC.Start();
    }
```

Запустив это приложение, можно обнаружить, что вывод на консоль перемешан из-за того, что потоки выполняются по очереди, и, не успев завершить вывод, оказываются прерванными другими потоками. Если перемешивания не происходит, можно поэкспериментировать со временем сна в некоторых пределах, от 0 до 10.

2.4.1. Синхронизация ключевым словом lock

Есть несколько способов синхронизации потоков.

Сначала синхронизируем вывод с помощью ключевого слова lock:

```
lock (this) {
    // синхронизированная область
}
```

В секцию lock нужно поместить цикл for метода DoWork целиком.

Результат будет зависеть от того, что будет блокирующим объектом.

Запускаем этот вариант блокировки, видим, что синхронизации нет.

Блокирующим объектом должен быть ссылочный тип.

Если заблокировать поток относительно класса потока, используя this, как в примере, тогда это должен быть один и тот же класс для всех потоков, а у нас классы разные, поэтому блокировки не происходит.

Создадим статический блокирующий объект (перед Main):

```
static System.Object locker = new System.Object();
```

Далее в классе потока объявим блокирующий объект:

```
internal class Worker {
    private volatile string name = "noname";
    private System.Object locker = null;
    public Worker(string name, object o) {
        this.name = name;
        locker = o;
    }
    public void DoWork() {
        lock (locker) {
            for (int i = 0; i < 5; i++) {
                Console.Write("Worker ");
                Thread.Sleep(0);
                Console.Write(name);
                Console.Write("-");
                Console.Write(i + " ");
                Thread.Sleep(0);
                Console.WriteLine();
            }
        }
    }
}
```

Соответственно, нужно изменить метод Main:

```
static void Main(string[] args) {
    Worker w1 = new Worker("A", locker);
    Thread wA = new Thread(new ThreadStart(w1.DoWork));
    Worker w2 = new Worker("B", locker);
    Thread wB = new Thread(new ThreadStart(w2.DoWork));
    Worker w3 = new Worker("C", locker);
    Thread wC = new Thread(new ThreadStart(w3.DoWork));
    . . .
}
```

Запускаем этот вариант блокировки, убеждаемся, что синхронизация достигнута.

2.4.2. Синхронизация монитором

Второй вариант — использовать монитор. В него нужно войти, выполнить работу и обязательно выйти (поэтому используется try-finally):

```
Monitor.Enter(this);
try {
    // синхронизированная область
} finally {
    Monitor.Exit(this);
}
```

На самом деле lock — это упрощенная версия монитора.

Чтобы применить монитор в нашем классе, нужно точно так же, как и в предыдущем случае, вместо this использовать объект locker:

```
public void DoWork() {
    Monitor.Enter(locker);
    try {
        for (int i = 0; i < 5; i++) {
            Console.Write("Worker ");
            Thread.Sleep(0);
            Console.Write(name);
            Console.Write("-");
            Console.Write(i + " ");
            Thread.Sleep(0);
            Console.WriteLine();
        }
    } finally {
        Monitor.Exit(locker);
    }
    //lock (locker) {
    //    for (int i = 0; i < 5; i++) {
    //        Console.Write("Worker ");
    //        Thread.Sleep(0);
    //        Console.Write(name);
    //        Console.Write("-");
    //        Console.Write(i + " ");
    //        Thread.Sleep(0);
    //        Console.WriteLine();
    //    }
    //}
}
```

Пробуем этот вариант, убеждаемся, что синхронизация достигается.

2.4.3. Синхронизация атрибутом Synchronized

Есть еще способ синхронизации, заключающийся в том, чтобы объявить метод потока синхронизируемым с помощью атрибута Synchronized.

Сначала нужно добавить using:

```
using System.Runtime.CompilerServices;
```

Перед методом DoWork описываем атрибут:

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void DoWork() {
    . . .
}
```

Имеющуюся синхронизацию монитором следует удалить, например, закомментировать.

Объект класса потока теперь может быть только единственным, без вариантов, поэтому следует изменить метод Main:

```
static void Main(string[] args) {
    //Worker w1 = new Worker("A", locker);
    //Thread wA = new Thread(new ThreadStart(w1.DoWork));
    //Worker w2 = new Worker("B", locker);
    //Thread wB = new Thread(new ThreadStart(w2.DoWork));
    //Worker w3 = new Worker("C", locker);
    //Thread wC = new Thread(new ThreadStart(w3.DoWork));
    Worker w = new Worker("X", locker);
    Thread wA = new Thread(new ThreadStart(w.DoWork));
    Thread wB = new Thread(new ThreadStart(w.DoWork));
    Thread wC = new Thread(new ThreadStart(w.DoWork));
    wA.Start();
    wB.Start();
    wC.Start();
}
```

Запускаем программу, убеждаемся, что синхронизация есть.

2.4.4. События и wait-методы

Создадим новое консольное приложение C#.

Название проекта SharpThreadsWait.

Здесь мы исследуем использование синхронизирующих событий. Есть два типа событий — сбрасываемые автоматически AutoResetEvent и вручную ManualResetEvent. Автоматически сбрасываемое событие переходит в несигнальное состояние после того, как ожидающий его поток возобновит свое исполнение.

События с ручным сбросом позволяют возобновить исполнение множества потоков, после чего они могут быть переведены в несигнальное состояние при помощи метода Reset (в сигнальное состояние событие переводится при помощи метода Set).

Поток переводится в состояние ожидания одним из методов пространства имен System.Threading.WaitHandle:

1) WaitOne — поток блокируется, пока одно событие не перейдет в сигнальное состояние;

2) WaitAny — поток блокируется, пока одно из событий не перейдет в сигнальное состояние;

3) WaitAll — поток блокируется, пока все события не перейдут в сигнальное состояние.

В следующем примере поток ожидает перевода события autoEvent с ручным сбросом в сигнальное состояние:

```
public class Program {
    // синхронизирующее событие
    private static AutoResetEvent autoEvent;
    . . .
```

Далее описывается функция потока:

```
    static void process() {
        Console.WriteLine("Second thread waits for event...");
        autoEvent.WaitOne();
        Console.WriteLine("Second thread resume and ends...");
    }
```

Метод Main:

```
    public static void Main() {
        Console.WriteLine("Main thread starts...");
        // создаем событие с начальным несигнальным состоянием
        autoEvent = new AutoResetEvent(false);
        // создаем поток и стартуем его
        Thread t = new Thread(process);
        t.Start();
        // приостанавливаем первичный поток на 3 секунды
        Thread.Sleep(3000);
        // переводим событие в сигнальное состояние
        Console.WriteLine("Main thread signals...");
        autoEvent.Set();
        // ждем воссоединения потоков и завершаем
        t.Join();
        Console.WriteLine("Main thread ends...");
    }
```

В заключение отметим, что различные свойства и методы потоков доступны через переменную потока, здесь они не рассматривались за недостатком времени. Например, каждому потоку можно задать имя, вместо того, чтобы присваивать имя переменной класса потока.

2.5. Контрольные вопросы и упражнения

1. Что может являться потоком?
2. Как создается и запускается поток?
3. Опишите методы синхронизации потоков.
4. Чем отличаются события с ручным и автоматическим сбросом?
5. Опишите wait-методы для потоков.