

**top**

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# Системное программирование

# Урок № 1

## Процессы. Многопоточность.

### Содержание

Введение .....	4
1. Основные сведения о процессах .....	8
2. Функции манипулирования процессами .....	24
3. Понятие дочернего процесса .....	30
4. Манипулирование дочерним процессом.....	34
5. Домен приложения .....	50
6. Использование доменов приложения .....	62
7. Многопоточность .....	77

8. Потоки .....	86
9. Практические примеры использования потоков .....	101
10. Домашнее задание .....	108

# Введение

Рассматривая проблему многозадачности, так или иначе, изложение фокусируется на таких конструкциях операционной системы, как процессы и потоки, которым и будет посвящён настоящий урок. Строго говоря, реализация многозадачности на физическом уровне, в большинстве случаев, сводятся к многопоточности.

В вычислительных машинах, работающих на 64-разрядной архитектуре, уже поддерживается возможность физического распараллеливания вычислений. Но говоря о 32-разрядных операционных системах, мы понимаем, что машинные инструкции могут исполняться процессором только строго последовательно.

Набор инструкций, организованный в последовательную процедуру, на программном уровне, называют *потоком исполнения*.

Но, хотя существование физически параллельных потоков исполнения в 32-разрядных операционных системах и не возможно, поддерживается логическая многозадачность. Операционная система "делит" (распределяет) процессорное время "поровну" между всеми заявленными потоками исполнения. Конечно же, мы понимаем, что процессорное время делиться не поровну, но об этом будет оговорено в главе, посвящённой процессам. И, так как на выполнение каждого потока исполнения выделяется сравнительно малая часть процессорного времени в единицу реального времени, то возникает иллюзия их параллельности.

Однако, на логическом уровне, реализация параллельно выполняющихся действий должна отражать привязку потока исполнения к некоторому приложению, которое бы отражало "сферу применения исполняемого алгоритма или совокупности алгоритмов". Другими словами, операционная система логически связывает поток исполнения с некоторой конструкцией, которая демонстрирует нам, что определённые, отдельно взятые, потоки исполнения связаны между собой по целям и задачам.

Обычно разделяют понятия многозадачности и многопоточности. Под многопоточностью понимают возможность параллельно выполнять некоторый исполняемый код, поскольку потоки являются единственной конструкцией, в которую допустимо загрузить исполняемый код и между которыми операционная система делит процессорное время, тогда как под многозадачностью понимают возможность параллельно исполнять множество приложений в рамках операционной системы.

Многозадачность реализуется путём создания конструкций операционной системы называемых процессами. Тогда как выполнение параллельных вычислений (операций с ресурсами) в рамках одного процесса называют многопоточностью.

Прежде чем перейти к изложению теоретического материала, считаем необходимым ввести несколько ключевых понятий.

Здесь и дальше (по мере необходимости), мы будем приводить вместе с русскими названиями понятий их

английский эквивалент, поскольку это упростит понимание структуры имён библиотечной системы типов, рассматриваемой в рамках текущего урока.

**Поток (thread)** – конструкция операционной системы, которая служит для размещения и выполнения исполняемого кода. Отсюда следует, что исполняемый код может быть расположен и, впоследствии, выполнен только внутри некоторого потока. На текущем этапе данного понятия будет достаточно для восприятия дальнейшего материала.

**Процесс (process)** – конструкция операционной системы, которая служит для ограничения доступа к ресурсам и адресному пространству, ассоциированным с некоторым приложением, а так же для размещения и исполнения потоков этого приложения.

**Адресное пространство (address space или memory scope)** – пространство адресов оперативной памяти, которое выделяется процессу, для хранения данных, используемых в рамках этого процесса.

**Сборка (assembly)** – запущенное приложение, JIT-скомпилированное и выгруженное в оперативную память. Как правило, сборка состоит из некоторого количества совместно исполняемых модулей, делящих одно и то же адресное пространство.

**Модуль (module)** – под модулем мы будем понимать некоторый исполняемый файл: \*.exe или \*.dll, который загружен, как составной элемент некоторой сборки.

**Ресурсы (resources)** – любые конструкции операционной системы, которые загружены процессом, в том числе и сборки.

*Приложение (application)* – на уровне использования это понятие можно отождествить с прикладным программным решением (или попросту программой).

# 1. Основные сведения о процессах

---

В самом упрощённом понимании, процесс – это выполняющаяся в операционной системе программа.

Однако процесс – это не только конструкция операционной системы. Её следует так же понимать и как категорию, которой операционная система (и, возможно, пользовательское приложение) манипулирует, для разделения и идентификации приложений в операционной системе. Так же процесс ограничивает доступ к адресному пространству и ресурсам, используемым запущенным в нём приложением и определяет уровень доступа приложения к ресурсам операционной системы.

Процесс наделён следующими *свойствами*:

- адресным пространством виртуальной памяти, в котором потоки процесса располагают используемые ими данные.

В управляемом коде все адреса процессозависимы, то есть, адрес, полученный, напрямую, в один процесс из другого процесса, не может быть использован. Для межпроцессного разыменования адреса необходимо использовать некоторый объект, который определит уровень и процедуру разыменования. Такие объекты называют прокси-объектами или объектами-посредниками (от англ. *proxy* – посредник);



- исполняемым кодом, помещённым в потоки из модулей, которые находятся в сборках, загружаемых процессом;
- открытыми дескрипторами системных объектов, необходимых для работы приложения;
- контекстом общей системы безопасности, в рамках которого запущен процесс. Контекст безопасности определяется политиками безопасности и разрешениями, применёнными к пользователю, от имени которого запущен процесс, и используется операционной системой для определения прав доступа к ресурсам, а так же прав на исполнение операций (действий);
- уникальным идентификатором, который служит для безусловной идентификации каждого отдельного процесса от остальных;
- переменными окружения, используемыми всеми потоками отдельно взятого процесса;
- классом приоритета, который, в совокупности с приоритетом каждого отдельного потока текущего процесса, используется операционно системой для определения необходимого процессорного времени, для каждого отдельного потока этого процесса;
- как минимум одним потоком исполнения, с которым связывают время жизни процесса.

Как вы понимаете, для всех конструкций, необходимых для получения информации о процессах, а так же управления ими, в библиотеке базовых классов (BCL) .NET Framework-a, описаны соответствующие типы данных. Необходимые для получения информации о

процессах и манипулирования ими типы данных локализованы в пространствах имён `System.Diagnostics` и `System.Reflection`. Мы рассмотрим их последовательно, по мере их появления в контексте изложения.

Класс `Process` объявлен в пространстве имён `System.Diagnostics` и служит для описания компонента операционной системы – **процесс**, а так же содержит объявление следующих компонентных свойств, методов и событий (мы будем рассматривать не все, а наиболее используемые и полезные в работе, компоненты классов).

### Свойства класса `Process`.

***BasePriority*** – возвращает базовый приоритет процесса – значение типа `int`, которое может приобретать следующие значения:

- 4 – `Idle` – низкий приоритет;
- 8 – `Normal` – нормальный приоритет;
- 13 – `High` – высокий приоритет;
- 24 – `RealTime` – приоритет реального времени.

***CanRaiseEvents*** – возвращает логическое значение, которое говорит о том, может ли процесс генерировать события релевантные процессу. Если свойство возвращает значение `false`, то установленные объекту обработчики событий отработывать не будут.

***EnableRaisingEvents*** – возвращает или устанавливает логическое значение, которое свидетельствует о том, может ли объект процесса генерировать релевантные ему события.

***ExitCode*** – возвращает значение, которое соответствующий процесс установил при его завершении.

***ExitTime*** – возвращает время, которое соответствует моменту завершения процесса, который описывается объектом данного типа.

***Handle*** – возвращает дескриптор соответствующего процесса.

***HandleCount*** – возвращает количество дескрипторов, открытых процессом.

***HasExited*** – возвращает логическое значение, которое свидетельствует о том, завершился ли процесс.

***Id*** – возвращает уникальный идентификатор процесса (Process Identifier или PID).

***MachineName*** – возвращает "имя" компьютера, на котором выполняется процесс, с которым ассоциирован текущий объект.

***MainModule*** – возвращает главный модуль процесса, с которым ассоциирован процесс.

***MainWindowHandle*** – возвращает дескриптор главного окна текущего процесса, в случае, если процесс оконный.

***MainWindowTitle*** – возвращает заголовок главного окна процесса, с которым ассоциирован текущий объект, в случае, если процесс оконный.

***Modules*** – возвращает коллекцию модулей (объект типа ProcessModuleCollection), загруженных процессом.

***PriorityBoostEnabled*** – возвращает логическое значение, которое свидетельствует о том, позволено ли процессу повышение приоритета, когда на его главное окно переводиться фокус.

***PriorityClass*** – возвращает или принимает объект перечисления ProcessPriorityClass, который определя-

ет класс приоритета процесса. Перечисление `ProcessPriorityClass` содержит следующие константы, которые соответствуют классам приоритета процесса:

- `Normal` – соответствует нормальному уровню приоритета. Нормальный уровень приоритета устанавливается всем процессам по умолчанию.
- `Idle` – указывает, что потоки процесса будут выполняться только тогда, когда система простаивает.
- `High` – указывает, что процесс выполняет очень значимые, критически важные операции, которые должны выполняться немедленно.
- `RealTime` – самый высокий приоритет, который может быть установлен в операционной системе.
- `BelowNormal` – приоритет, находящийся между `Normal` и `Idle`.
- `AboveNormal` – приоритет, который находится между `Normal` и `High`.

***ProcessName*** – возвращает имя процесса.

***Responding*** – возвращает логическое значение, которое говорит о том, "отвечает ли" пользовательский интерфейс процесса на действия пользователя.

***StandardError*** – возвращает поток, который используется для чтения вывода ошибок возникающих в приложении.

***StandardInput*** – возвращает поток ввода приложения, выполняющегося в рамках процесса.

***StandardOutput*** – возвращает поток вывода приложения, выполняющегося в рамках процесса.

***StartInfo*** – возвращает или устанавливает объект

класса `ProcessStartInfo`, который содержит набор значений, используемых для запуска процесса. Это свойство передаётся как аргумент при вызове метода `Start` класса `Process` (метода, запускающего процесс). Перечислим наиболее важные свойства этого класса:

- `Arguments` – устанавливает аргументы командной строки которые должно принять приложение, запускаемое в рамках процесса.
- `CreateNoWindow` – возвращает или устанавливает логическое значение, которое говорит о том, запускается ли процесс как оконный, то есть создаёт ли процесс окно при запуске.
- `EnvironmentVariables` – возвращает строковый словарь (объект класса `StringDictionary`), содержащий переменные окружения.
- `FileName` – возвращает или устанавливает путь к приложению (документу), которое запускается в рамках процесса.
- `LoadUserProfile` – возвращает и устанавливает логическое значение, которое указывает должен ли при запуске процесса пользовательский профиль быть загружен из реестра.
- `Password` – возвращает или устанавливает пользовательский пароль.
- `UserName` – возвращает или устанавливает имя пользователя запускающего процесс.
- `WindowStyle` – возвращает и устанавливает оконные стили, которые устанавливаются главному окну во время запуска процесса.

- **WorkingDirectory** – возвращает и устанавливает рабочую директорию процесса.

**Threads** – возвращает множество потоков, исполняющихся в рамках текущего процесса.

### **Методы класса Process.**

**Close** – освобождает все ресурсы, которые использовались ассоциированным процессом.

**CloseMainWindow** – инициирует закрытие главного окна процесса.

**GetCurrentProcess** – статический метод класса, который возвращает объект процесса, в котором выполняется поток, вызвавший данный метод.

**GetProcessById** – возвращает процесс, PID которого равен значению, переданному в качестве аргумента.

**GetProcesses** – возвращает массив объектов класса Process, который соответствует множеству всех процессов, запущенных в операционной системе.

**GetProcessesByName** – возвращает множество процессов по их имени.

**Kill** – немедленно прекращает выполнение процесса.

**OnExited** – инициирует событие Exited.

**Refresh** – отменяет все изменения, которые были внесены в компонент процесса, с момента его запуска.

**Start** – запускает процесс на выполнение.

**WaitForExit** – устанавливает интервал времени, который вызывающий этот метод поток будет ожидать закрытия текущего процесса.

### **События класса Process.**

**ErrorDataReceived** – инициируется, когда приложение, запущенное в рамках процесса, записывает данные в стандартный поток ошибок процесса.

*Exited* – инициируется, когда процесс завершается.

*OutputDataReceived* – инициируется, когда приложение, запущенное в рамках процесса, записывает данные в стандартный поток вывода процесса.

Тип данных *Assembly* характеризует объект сборки, загруженной в рамках процесса, и описан в пространстве имён *System.Reflection*. Тип данных *Assembly* содержит объявление следующий компонентных элементов.

#### **Свойства:**

*EntryPoint* – возвращает точку входа для текущей сборки;

*Evidence* – возвращает объект класса *Evidence*, который определяет разрешения политики прав безопасности, предоставленные текущей сборке;

*FullName* – возвращает имя сборки;

*GlobalAssemblyCache* – возвращает логическое значение, которое свидетельствует о том, была ли сборка загружена из глобального кэша сборок;

*ImageRuntimeVersion* – возвращает строку, которая содержит информацию о версии единой среды исполнения;

*Location* – возвращает местонахождение файла, содержащего манифест текущей сборки;

*ManifestModule* – возвращает объект класса *Module* (модуль сборки), который содержит манифест текущей сборки;

*ReflectionOnly* – возвращает логическое значение, которое говорит о том, была ли текущая сборка загружена в контекст, который предназначен только для "отражения", но не для исполнения.

**Методы:**

*CreateInstance* – создаёт экземпляр типа данных, описанного в рамках текущей сборки, по его имени;

*GetAssembly* – возвращает загруженную в текущий момент времени сборку, в которой объявляется класс, переданный в качестве параметра;

*GetCallingAssembly* – возвращает сборку, в которой расположен метод, вызывающий текущий метод;

*GetCustomAttributes* – возвращает атрибуты, установленные для этой сборки;

*GetExecutingAssembly* – возвращает сборку, которая содержит код, исполняемый в момент вызова текущего метода;

*GetExportedTypes* – возвращает типы данных, которые определены в текущей сборке, и в данный момент времени открыты и видимы за её пределами;

*GetLoadedModules* – возвращает массив всех модулей, загруженных текущей сборкой;

*GetModule* – возвращает конкретный модуль сборки по его имени;

*GetModules* – возвращает массив всех модулей текущей сборки;

*GetName* – возвращает объект AssemblyName, который описывает текущую сборку;

*GetReferencedAssemblies* – возвращает множество всех сборок, на которые ссылается текущая сборка;

*GetType* – возвращает объект Type, описывающий конкретный тип данных, объявленный в текущей сборке, по его имени;



***GetTypes*** – возвращает множество всех типов данных описанных в текущей сборке;

***IsDefined*** – возвращает логическое значение, которое указывает, применён ли, переданный в качестве параметра, атрибут к текущей сборке;

***Load*** – выполняет загрузку. Выполнять загрузку сборки можно как на основе длинного имени сборки, переданного через строку, так и на основе объекта *AssemblyName*, который абсолютно точно идентифицирует сборку;

***LoadFile*** – статический метод, который выполняет загрузку сборки из файла и возвращает объект загруженной сборки;

***LoadFrom*** – статический метод, который загружает сборку на основании заданного имени сборки или пути к файлу, содержащему сборку, и возвращает объект класса *Assembly*, описывающего загруженную сборку;

***LoadModule*** – загружает модуль в текущую сборку;

***ReflectionOnlyLoad*** – загружает сборку в контексте, предназначенный только для отражения. Таким образом, исполняемый код этой сборки не может быть выполнен.

Класс *Module* описывает модуль некоторой сборки, который, как правило, ассоциируется с некоторым исполняемым файлом (\*.exe или \*.dll). Класс описан в пространстве имён *System.Reflection*. Класс содержит описание следующих компонентов.

**Свойства:**

*Assembly* – возвращает объект *Assembly*, описывающий сборку, в которую загружен текущий модуль;

*FullyQualifiedName* – возвращает полное название модуля, включая путь, по которому он расположен;

*ModuleHandle* – возвращает дескриптор, описывающий модуль; Дескриптор можно использовать при вызове Win32 API методов, загруженных в приложение;

*ModuleVersionId* – возвращает идентификатор, определяющий версию модуля;

*Name* – возвращает имя модуля.

**Методы:**

*Invoke* – выполняем некоторый метод, объявленный в текущем модуле;

*GetCustomAttributes* – возвращает массив объектов *AttributeInfo*, соответствующий множеству всех атрибутов текущего модуля;

*GetField* – возвращает объект *FieldInfo*, описывающий поле некоторого класса, объявленного в текущем модуле;

*GetFields* – принимает имя класса и возвращает массив объектов *FieldInfo*, соответствующий множеству всех полей этого класса; Если класс не объявлен в текущем модуле, то в качестве результата мы получим ссылку со значением *null*;

*GetMethod* – возвращает объект класса *MethodInfo*, описывающий метод, переданный в качестве параметра; Само собой, что метод с переданным именем должен быть описан в текущем модуле; Если метод не объявлен в текущем модуле, то в качестве результата мы получим ссылку со значением *null*;

***GetMethods*** – возвращает массив объектов MethodInfo, который соответствует всему множеству методов описанных в рамках текущего модуля;

***GetType*** – возвращает объект класса Type, описывающий тип данных, объявленный в текущем модуле;

***GetTypes*** – возвращает массив объектов класса Type, соответствующий множеству всех типов данных, объявленных в текущем модуле;

***IsDefined*** – возвращает логическое значение, которое говорит, установлен ли для модуля атрибут, переданный в качестве параметра;

***IsResource*** – возвращает логическое значение, которое свидетельствует о том, является ли текущий модуль ресурсом.

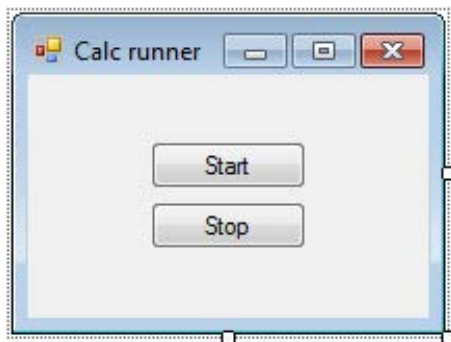
### Создание простейшего приложения, "порождающего" процесс

Теперь мы рассмотрим использование объекта типа данных Process, в контексте задач манипулирования процессами. Мы создадим простейшее оконное приложение, которое будет запускать приложение калькулятор (файл "calc.exe") и, впоследствии, останавливать его.

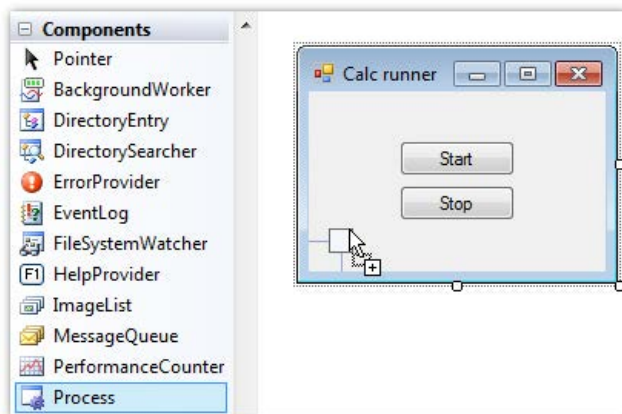
Далее перечислены действия по созданию приложения:

1. В диалоговом окне "Новый проект" создайте приложение, используя шаблон Windows Forms Application в группе шаблонов для языка программирования Visual C#.
2. В открытом конструкторе Form1 выберите вкладку Windows Forms панели элементов и добавьте

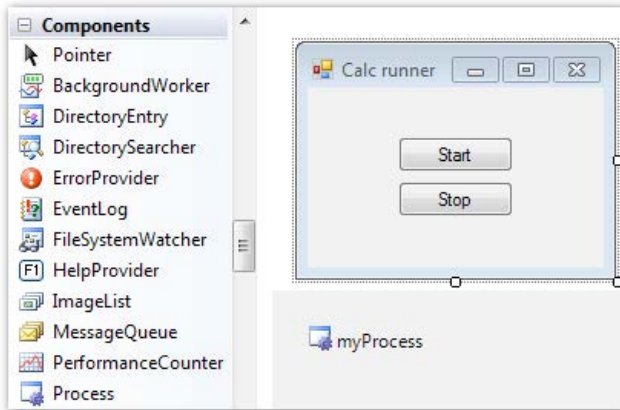
в форму две кнопки: Start и Stop, которые, соответственно, будут запускать и останавливать процесс.



3. Присвойте кнопкам осмысленные имена и текст, как показано на картинке, а так же создайте для них обработчики события "Click".
4. Откройте вкладку Компоненты панели элементов и перетащите в область конструктора экземпляр компонента Process.



5. После перетаскивания компонент Process появиться в окне редактора на нижней панели оконного дизайнера, как это показано на рисунке ниже.



6. По умолчанию компонент имеет название process1 – назовите его myProcess.
7. Для того, чтобы компонент знал какую сборку необходимо запустить в рамках вновь созданного процесса, необходимо установить значение свойства StartInfo компонента my Process. Свойство StartInfo принимает объект типа ProcessStartInfo, описанном в пространстве имён System.Diagnostics. Следовательно, сперва необходимо подключить пространство имён в проект, используя директиву using.

```
using System.Diagnostics;
```

Проинициализировать свойство `StartInfo` можно в конструкторе формы, после вызова метода `InitializeComponents`.

```
public CalcRunner()
{
    InitializeComponent();
    myProcess.StartInfo = new System.Diagnostics.
    ProcessStartInfo("calc.exe");
}
```

Компонент `Process` ищет файл, по относительному пути от домашней директории текущего процесса и, если файл не найден, то он "пробует" найти файл в системной директории. По умолчанию – это директория `C:\WINDOWS\System32`. Таким образом, в нашем случае, в результате вызова метода `Start` компонента `myProcess` будет запущено приложение калькулятор, в рамках отдельного процесса.

8. Для того, чтобы запустить процесс, необходимо в обработчике события `Click` кнопки `Start` вызвать компонентный метод `Start` объекта `myProcess`.

```
private void Start_Click(object sender, EventArgs e)
{
    myProcess.Start();
}
```

Для того, чтобы остановить процесс, достаточно вызвать компонентный метод `CloseMainWindow` объекта `myProcess`, который приведёт к закрытию главного окна открытого в процессе приложения, а значит основ-

ной поток этого приложения перейдёт к точке выхода и завершиться, что, в свою очередь, приведёт к закрытию процесса. Однако, после закрытия, необходимо очистить ресурсы, зарезервированные процессом. Для этой цели используется компонентный метод `Close` компонента `Process`. Закрытие процесса мы реализуем в обработчике события `Click` кнопки `Stop`.

```
private void Stop_Click(object sender, EventArgs e)
{
    myProcess.CloseMainWindow();
    myProcess.Close();
}
```

## 2. Функции манипулирования процессами

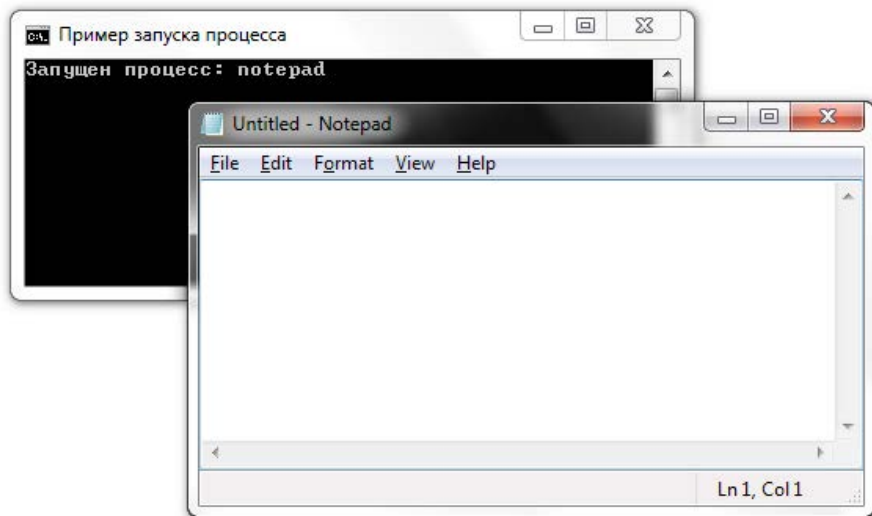
Приведём несколько примеров, демонстрирующих использование описанных выше типов данных.

Первый будет заключаться в запуске процесса на основании некоторого исполняемого файла. Сперва необходимо создать объект типа `Process`. Затем нужно передать в свойство `FileName` свойства `StartInfo` созданного объекта путь к файлу, который будет запущен в рамках, созданного нами процесса.

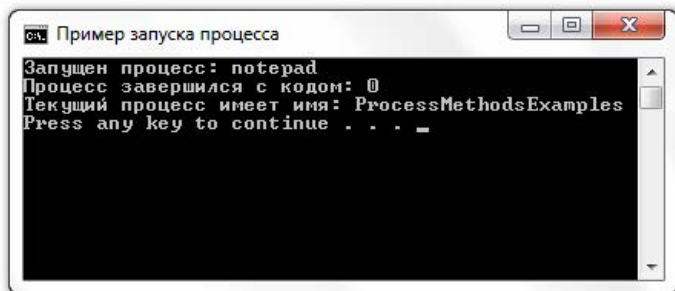
```
//Объявляем объект класса Process
Process proc = new Process();
//устанавливаем имя файла, который будет запущен в
//рамках процесса
proc.StartInfo.FileName = "notepad.exe";
//запускаем процесс
proc.Start();
//выводим имя процесса
Console.WriteLine("Запущен процесс: " + proc.
ProcessName);
//ожидаем закрытия процесса
proc.WaitForExit();
//выводим код, с которым завершился процесс
Console.WriteLine("Процесс завершился с кодом: " +
proc.ExitCode);
//выводим имя текущего процесса
Console.WriteLine("Текущий процесс имеет имя: "+
Process.GetCurrentProcess().ProcessName);
```



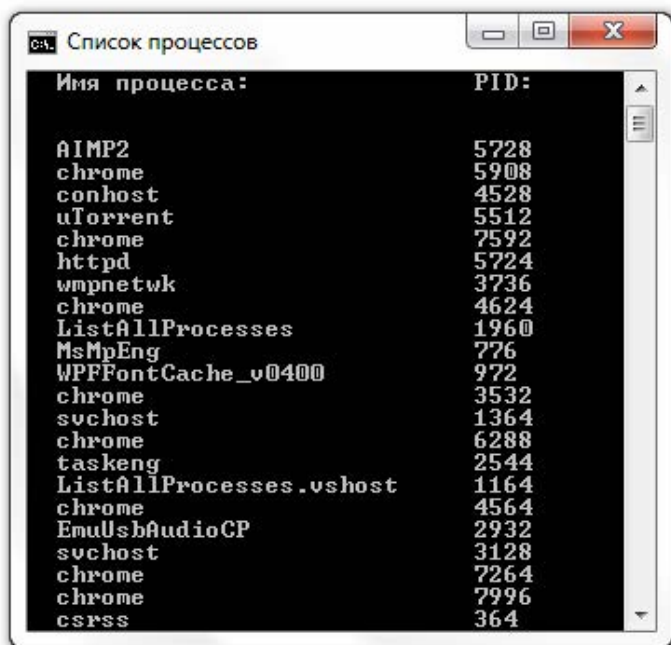
В представленном примере мы также вызываем метод `WaitForExit`, который заставляет поток, осуществивший вызов этого метода, ждать завершения процесса, от которого этот метод был вызван. Поэтому, в настоящем примере, завершение приложения произойдёт только после того, как мы закроем появившееся окно блокнота.



После того, как будет закрыто окно блокнота, выводится код, с которым завершился процесс, а так же имя текущего процесса.



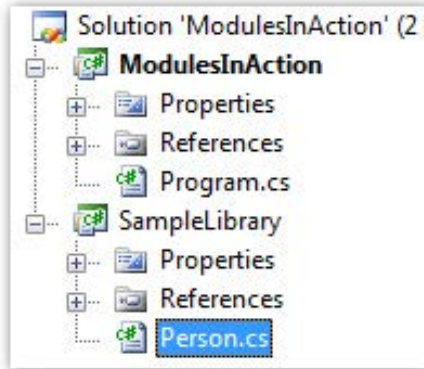
Следующий пример выводит список процессов, запущенных в операционной системе. Список процессов мы получаем посредством вызова статического метода класса `Process` `GetProcesses`, который, как уже было сказано, возвращает массив объектов `Process`, соответствующий множеству всех процессов, запущенных в операционной системе. Текст программы представлен ниже.



```
//устанавливаем заголовок консоли
Console.Title = "Список процессов";
//изменяем размер буфера консоли и окна на
//необходимые нам
Console.WindowWidth = 40;
Console.BufferWidth = 40;
```

```
//получаем список процессов
Process[] processes = Process.GetProcesses();
//выводим заголовок
Console.WriteLine("    {0,-28}{1,-10}", "Имя
процесса:", "PID:");
//для каждого процесса выводим имя и PID
foreach (Process p in processes)
    Console.Write("    {0,-28}{1,-10}", p.ProcessName, p.Id);
```

Следующий пример будет демонстрировать взаимодействие с объектами объекты Assembly и Module.



В рамках решения мы создадим два проекта: один представляет собой библиотеку, которую мы будем динамически загружать в и получать из неё тип данных и вызывать методы созданного объекта.

Текст библиотеки представлен ниже. Он описывает понятие личность (класс Person), а также понятие – сотрудник (класс Employee, производный от Person). В классе Person описывается метод Print, который осуществляет форматированный вывод информации, содержащейся в классе.

```

public enum PersonMaritalStatus
{
    Married,
    Single
}
[Serializable]
public class Person
{
    string Name;
    string LastName;
    int Age;
    PersonMaritalStatus MaritsalStatus;
    public Person(string Name, string Lastname,
                  int Age)
    {
        this.Name = Name;
        this.LastName = Lastname;
        this.Age = Age;
        this.MaritsalStatus =
            PersonMaritalStatus.Single;
    }
    public void Print()
    {
        Console.WriteLine("Person:\nName: " +
                          Name + "\nLastname: " +
                          LastName + "\nAge: " + Age);
    }
}
public class Employee : Person{
    string Position;
    decimal Salary;
    public Employee(string Name, string Lastname,
                  int Age,
string Position, decimal Salary): base(Name,
                  Lastname,Age)
    {
        this.Position = Position;
    }
}

```

```

        this.Salary = Salary;
    }
}

```

Получившийся dll-файл необходимо положить в директорию bin/debug проекта ModuleInAction. Этот проект имеет текст:

```

//загружаем сборку
Assembly asm = Assembly.Load(AssemblyName.
GetAssemblyName("SampleLibrary.dll"));
//получаем необходимый модуль этой сборки
Module mod = asm.GetModule("SampleLibrary.dll");
//выводим список типов данных, объявленный в текущем модуле
Console.WriteLine("Объявленные типы данных:");
foreach (Type t in mod.GetTypes())
    Console.WriteLine(t.FullName);
//получаем тип данных из сборки
Type Person = mod.GetType("SampleLibrary.Person") as
Type;
//создаём объект полученного типа данных
object person = Activator.CreateInstance(Person,
    new object[]{"Иван", "Иванов", 30});
//вызываем метод Print от созданного объекта
Person.GetMethod("Print").Invoke(person, null);

```

В результате запуска данного приложения мы должны получить вывод, который показан на изображении справа. Этот вывод говорит нам о том. Что в модуле объявлены три типа данных, а так же представляет инкапсулированные в созданном нами объекте типа Person поля.

### 3. Понятие дочернего процесса

---

Как правило, одно приложение ограничивается рамками одного процесса. Но иногда возникают ситуации, когда приложению необходимо запустить дочерний модуль в отдельном процессе. Примером такой ситуации может быть необходимость запустить системный калькулятор или другой системный модуль, как, например, делает программный комплекс "1С", или возможна ситуация, когда одна из функций приложения состоит в запуске других приложений, как у файлового менеджера (проводник ОС Windows или Total Commander). В таком случае обычно говорят, что некоторый процесс порождает дочерние процессы.

*Дочерним* называется *процесс*, запущенный из другого процесса, который, в свою очередь, становится для него родительским. Другими словами, если процесс "X" запускает процесс "Y", то процесс "Y" становится дочерним для "X", а "X" – родительским для "Y".

Подобный способ запуска дочерних приложений, которые логически можно считать модулями некоторого программного комплекса, имеет и положительные и отрицательные стороны.

Положительные моменты заключаются в том, что если завершается один из процессов, независимо от того, дочерний он или родительский, остальные продолжают работать, потому, что физически они никак не

связаны и не зависят друг от друга в смысле технологии их запуска и исполнения. Однако один из процессов может ожидать от другого некоторой информации, необходимой для его функционирования. В таком случае процесс продолжит работать, но перестанет выполнять свою функцию. Конечно, всё зависит от того, обрабатывается ли ошибка получения некорректных данных в коде. Если нет, то и этот процесс может прекратить выполнение, поскольку будет иметь место ошибка времени исполнения.

Так же, поскольку исполняемый код отдельных процессов не связан, то нет опасности, что ошибки исполнения в одном процессе, приведут к закрытию другого, что вносит некоторую степень стабильности программного комплекса. Таким образом, например, устроен браузер Google Chrome, у которого каждая вкладка представлена отдельным процессом, что позволяет всему браузеру продолжать работать даже тогда, когда на одной из его вкладок возникает необрабатываемая ошибка.

Но, так как адресное пространство процесса не доступно напрямую из другого процесса, поскольку, как уже упоминалось, адресация процессо-зависима, то межпроцессное взаимодействие составляет некоторое неудобство, что можно считать отрицательным моментом в смысле разработки.

Достаточно редко возникает ситуация, в которой необходимо осуществлять межпроцессный вызов методов. Но, достаточно часто возникает необходимость обмениваться данными между процессами. В операционной системе Windows предусмотрены способы межпро-

цессного взаимодействия: например, DDE (Dynamic Data Exchange – динамический обмен данными) или MMF (memory-mapped-files или файлы, проецируемые в память).

DDE и MMF представляют собой программные интерфейсы, которые реализованы на Win32 API. К сожалению, нет "управляемого варианта" библиотек DDE Management Library и Memory-Mapped-Files Management, как, например, System.Management для WMI интерфейса. Поэтому, в любом случае, при использовании DDE-интерфейса или просто используя систему сообщений операционной системы, взаимодействие сводится к вызову неуправляемых API функций (как правило использование SendMessage, в большинстве случаев, проще).

Для того, чтобы использовать некоторую системную функцию (функцию Win32 API), её необходимо выгрузить в приложение. Одним из способов является использование атрибута DllImport при описании метода класса. Например, функцию SendMessage мы, в дальнейших примерах, будем выгружать следующим образом:

```
[DllImport("user32.dll")]  
public static extern IntPtr SendMessage(IntPtr hwnd,  
uint Msg, int wParam,  
[MarshalAs(UnmanagedType.LPStr)]string lParam);
```

В качестве параметра атрибута DllImport мы передаём имя библиотеки, в которой необходимо искать функ-

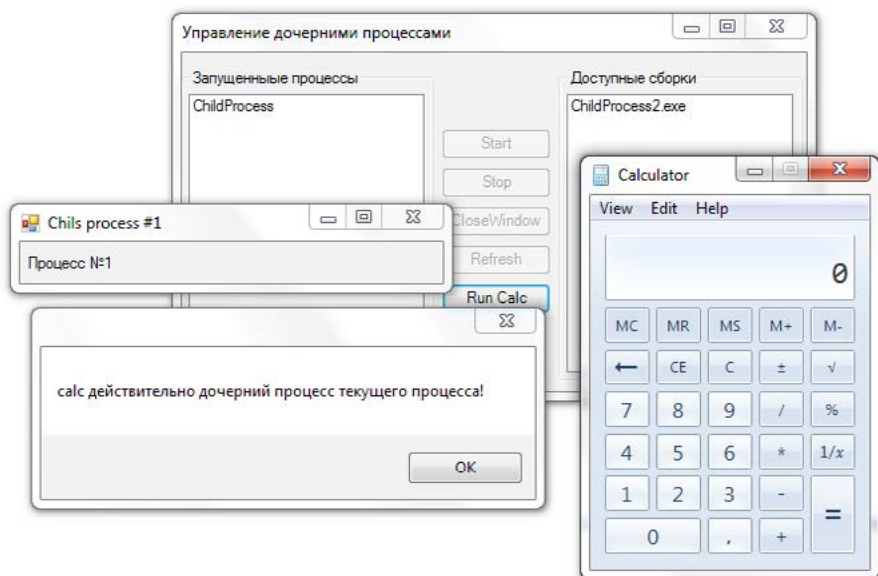


цию (получить информацию о том, в какой библиотеке расположена та или иная функция можно в официальной документации).

Так как в API не объявлен тип данных `string`, а также потому, что `string` в C# имеет несколько иную структуру, нежели оригинальный `LPSTR`, то мы используем атрибут `MarshalAs` для преобразования тип данных `LPSTR` к `string`.

## 4. Манипулирование дочерним процессом

Манипулирование дочерними процессами некоторого основного процесса мы рассмотрим на примере, в котором созданное нами оконное приложение (запущенное в отдельном процессе), будет вызывать дочерние процессы на базе исполняемых файлов, находящихся в домашней директории, и запускать системное приложение "calc.exe". А так же будет проверять и подтверждать, что запущенный процесс действительно является дочерним по отношению к текущему, в котором будет запущено наше приложение. Ниже показано, как будет выглядеть законченное приложение.

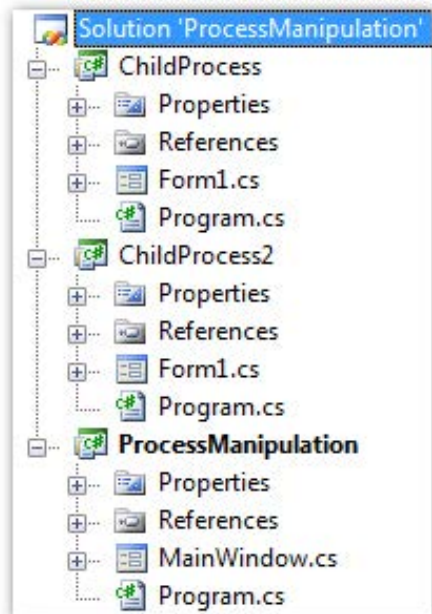


Для начала, необходимо создать проект по шаблону `WindowsFormsApplication` (оконное приложение `Windows Forms`), и к получившемуся решению добавить два проекта, которые будут запускаться основным приложением в качестве дочерних процессов.

Проекты, которые будут запускаться как дочерние процессы можно назвать `ChildProcess` и `ChildProcess2`. После компиляции решения исполняемые файлы проектов `ChildProcess` и `ChildProcess2` необходимо скопировать в директорию `./bin/debug` проекта `Process Manipulation`.

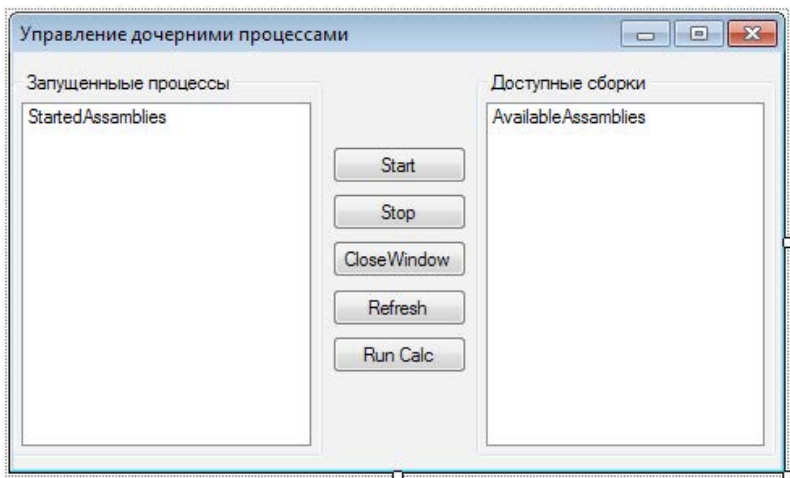
Окна приложений необходимо настроить следующим образом:

1. На окно основного проекта необходимо добавить два элемента управления `ListBox` – один предназначен для хранения и представления пользователю списка доступных для запуска приложений и будет называться `AvailableAssemblies`, а второй – для хранения списка запущенных приложений и будет называться `StartedAssemblies`.

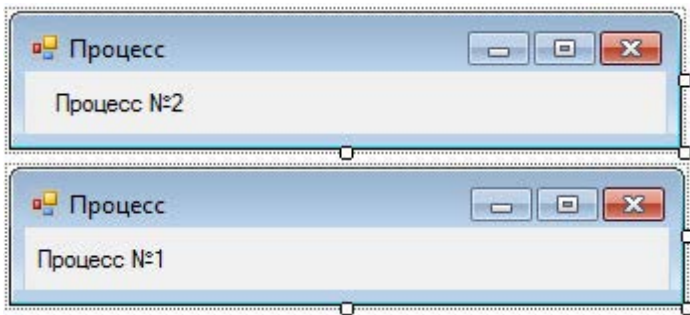


2. На это же окно необходимо добавить пять кнопок:
- a. Start, по нажатию на которую в отдельном процессе будет запускаться приложение, выбранное в списке доступных приложений;
  - b. Stop, по нажатию на которую будет останавливаться приложение, выбранное в списке запущенных приложений;
  - c. CloseWindow, по нажатию на которую будет закрываться главное окно приложения, выбранного в списке запущенных приложений;
  - d. Refresh, по нажатию на которую будет происходить обновление процесса, в котором запущено приложение, выбранное в списке запущенных приложений;
  - e. RunCalc, по нажатию на которую будет происходить запуск калькулятора.

Получившееся окно должно выглядеть подобным образом:



3. Свойство `Enabled` всех кнопок необходимо установить в значение `false`, чтобы впоследствии они включались только в те ситуациях, когда действие, которое они запускают, может быть выполнено.
4. Окна проектов `ChildProcess` и `ChildProcess2` необходимо настроить так, как показано ниже, то есть чтобы в них был элемент управления `Label`, с текстом, указывающим, которое из приложений запущено.



Далее последует описание кода основного проекта.

Сперва необходимо указать директивы `using` для нескольких пространств имён, типы данных которых будут нами использованы в проекте:

```
using System.Diagnostics;  
using System.Runtime.InteropServices;  
using System.IO;  
using System.Reflection;  
using System.Management;
```

Затем в класс основного окна приложения мы добавим объявление нескольких полей и методов:

```
//константа, идентифицирующая сообщение WM_SETTEXT
const uint WM_SETTEXT = 0x0C;
//импортируем функцию SendMessage из библиотеки
//user32.dll
[DllImport("user32.dll")]
public static extern IntPtr SendMessage(IntPtr hwnd,
uint Msg, int wParam,
    [MarshalAs(UnmanagedType.LPStr)]string lParam);
/*список, в котором будут храниться объекты,
 * описывающие дочерние процессы приложения*/
List<Process> Processes = new List<Process>();
/*счётчик запущенных процессов*/
int Counter = 0;
```

При объявлении метода `SendMessage` мы указываем атрибут `DllImport` для того, чтобы среда исполнения знала, что этот метод необходимо динамически выгрузить из библиотеки `user32.dll`, потому, что для отправки системного сообщения .NET Framework не имеет стандартных библиотечных методов.

Далее мы описываем метод, который будет получать из домашней директории проекта список приложений, которые можно запустить в качестве дочерних процессов.

```

/*метод, загружающий доступные исполняемые
 * файлы из домашней директории проекта*/
void LoadAvailableAssemblies()
{
    //название файла сборки текущего приложения
    string except = new FileInfo(Application.
ExecutablePath).Name;
    //получаем название файла без расширения
    except = except.Substring(0, except.
IndexOf("."));
    //получаем все *.exe файлы из домашней
    //директории
    string[] files = Directory.GetFiles(Application.
StartupPath, "*.exe");
    foreach (var file in files)
    {
        //получаем имя файла

        string fileName = new FileInfo(file).Name;

        /*если имя файла не содержит имени исполняемого
        *файла проекта, то оно добавляется в список*/

        if (fileName.IndexOf(except) == -1)

            AvailableAssemblies.Items.Add(fileName);
    }
}

```

Описанный метод будет вызван нами в конструкторе окна:

```

public MainWindow()
{
    InitializeComponent();
    LoadAvailableAssemblies();
}

```

Далее нам необходимо описать метод, который будет выполнять запуск приложения в дочернем процессе:

```

/*метод, запускающий процесс на исполнение и
 * сохраняющий объект, который его описывает*/
void RunProcess(string AssamblyName)
{
    //запускаем процесс на соновании исполняемого
    //файла
    Process proc = Process.Start(AssamblyName);
    //добавляем процесс в список
    Processes.Add(proc);
    /*проверяем, стал ли созданный процесс дочерним,
     *по отношению к текущему и, если стал, выводим
     *MessageBox*/
    if (Process.GetCurrentProcess().Id ==
        GetParentProcessId(proc.Id))

        MessageBox.Show(proc.ProcessName +

            " действительно дочерний процесс текущего
            процесса!");

    /*указываем, что процесс должен генерировать
     *события*/
    proc.EnableRaisingEvents = true;
    //добавляем обработчик на событие завершения
    //процесса
    proc.Exited += proc_Exited;
    /*устанавливаем новый текст главному окну
     *дочернего процесса*/
    SetChildWindowText(proc.MainWindowHandle, "Child
process #" + (++Counter));
    /*проверяем, запускали ли мы экземпляр такого
     *приложения и, если нет, то добавляем в список
     *запущенных приложений*/
    if (!StartedAssemblies.Items.Contains(proc.
        ProcessName))

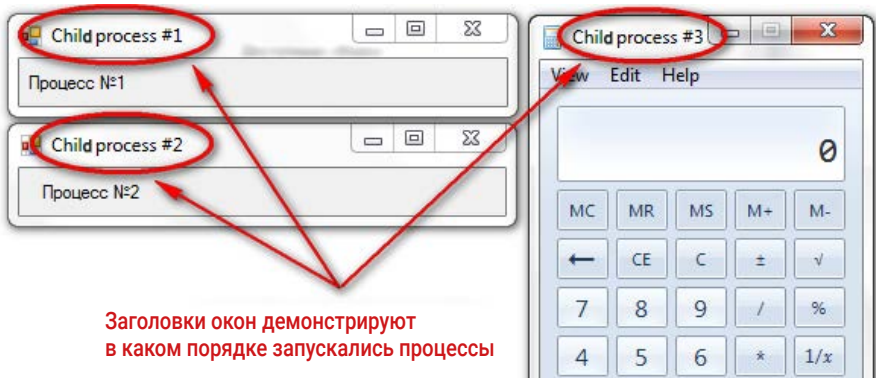
        StartedAssemblies.Items.Add(proc.ProcessName);
    /*убираем приложение из списка доступных
     *приложений*/
    AvailableAssemblies.Items.
        Remove(AvailableAssemblies.SelectedItem);
}

```



При запуске процесса, мы изменяем заголовок его основного окна на "Child process #" и прибавляем порядковый номер, руководствуясь текущим значением счётчика. Чтобы видеть в каком порядке были запущены процессы. В свою очередь это один из способов манипулирования дочерними процессами – изменение значений их переменных, а так же управление их поведением поведения.

Нижеприведённый рисунок демонстрирует – как будут выглядеть окна с изменёнными заголовками.



Для изменения текста заголовков нам необходим некоторый метод, который будет обёртывать вызов функции SendMessage:

```
/*метод обёртывания для отправки сообщения
 *WM_SETTEXT*/
void SetChildWindowText(IntPtr Handle, string text)
{
    SendMessage(Handle, WM_SETTEXT, 0, text);
}
```

Мы не можем напрямую получить доступ из одного процесса к адресному пространству другого процесса. Поэтому мы не можем просто изменить значение переменной, хранящей текст заголовка главного окна дочернего процесса. В качестве решения этой проблемы, мы можем сказать окну, что хотим, чтобы оно изменило свой текст, посредством отправки сообщения WM\_SETTEXT.

Для того, чтобы иметь такую возможность, мы выгрузили из библиотеки user32.dll Win32API функцию SendMessage, которая отправляет сообщение.

Далее мы описываем метод, который будет возвращать PID родительского процесса, некоторого процесса, переданного ему в качестве аргумента:

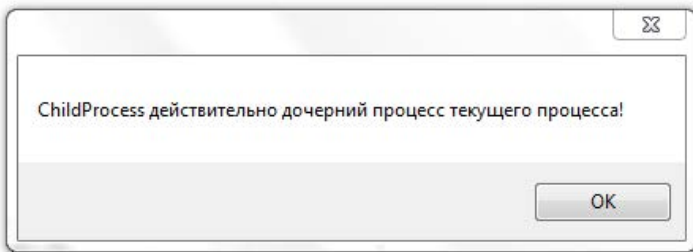
```
/*метод, получающий PID родительского процесса
*(использует WMI)*/
int GetParentProcessId(int Id)
{
    int parentId = 0;
    using (ManagementObject obj =
        new ManagementObject("win32_process.handle=" +
                               Id.ToString()))
    {
        obj.Get();

        parentId = Convert.
            ToInt32(obj["ParentProcessId"]);
    }
    return parentId;
}
```

В этом методе нами был использован тип данных ManagementObject, который представляет собой экземпляр WMI (Windows Management Instrumentation

– Инструментарий управления Windows). WMI не является объектом изложения текущего урока, тем более, что WMI – это прикладной интерфейс настройки операционной системы Windows, и полная документация по его структуре доступна в MSDN. Достаточно будет сказать, что посредством этого инструментария мы можем получить PID родительского процесса некоторого, произвольно выбранного нами процесса, по его PID. Для этого мы создаём объект класса ManagementObject на основе WMI класса win32\_process, с указанием поля handle. Таким образом, будет создан объект управления процессом с переданным PID. После чего становится доступно поле ParentProcessId.

Описанный выше метод вызывается при запуске дочернего процесса и значение, которое он возвращает, сравнивается с PID текущего (основного) процесса. Если значения совпадают, то мы делаем вывод, что переданный в качестве аргумента процесс является дочерним процессом текущего процесса и для подтверждения этого выводим MessageBox, с соответствующим сообщением, как показано на приведённом ниже рисунке.



Описываем обработчик события Exited класса Process:

```

/*обработчик события Exited класса Process*/
void proc_Exited(object sender, EventArgs e)
{
    Process proc = sender as Process;
    //убираем процесс из списка запущенных
    //приложений
    StartedAssemblies.Items.Remove(proc.ProcessName);
    //добавляем процесс в список доступных
    //приложений
    AvailableAssemblies.Items.Add(proc.ProcessName);
    //убираем процесс из списка дочерних процессов
    Processes.Remove(proc);
    //уменьшаем счётчик дочерних процессов на 1
    Counter--;
    int index = 0;
    /*меняем текст для главных окон всех дочерних
    *процессов*/
    foreach (var p in Processes)

        SetChildWindowText(p.MainWindowHandle, "Child
                                                                    process #" + ++index);
}

```

Как реакцию на завершение некоторого процесса, мы выполняем следующий набор действий:

1. убираем название процесса из списка запущенных приложений;
2. добавляем название закрытого приложения в список доступных приложений;
3. убираем процесс, в котором выполнялось приложение, из списка дочерних процессов, а так же для главных окон всех дочерних процессов меняем заголовков, чтобы они имели правильный порядковый номер. Проверить это можно запусти несколько приложений, а потом закрыв одно на выбор (ну, естественно, не последнее, иначе не будет видно никаких изменений).

Так как мы будем выполнять над процессом действия, основываясь на его имени, а одноимённых процессов в операционной системе может быть запущено любое количество, поскольку система идентифицирует процессы основываясь на их PID, то до того, как начать описывать обработчики событий нажатия на кнопки, созданные нами в основном окне, нам понадобится описать метод, который будет получать все процессы с некоторым именем, проходить по списку процессов и для каждого выполнять некоторое действие, определённое методом, на который будет ссылаться делегат, переданный в качестве аргумента:

```
//объявление делегата, принимающего параметр типа
//Process
delegate void ProcessDelegate(Process proc);
/*метод, который выполняет проход по всем дочерним
*процессам с заданным именем и выполняющий для этих
*процессов заданный делегатом метод*/
void ExecuteOnProcessesByName(string ProcessName,
                               ProcessDelegate func)
{
    /*получаем список, запущенных в операционной
    *системе процессов*/
    Process[] processes = Process.
        GetProcessesByName(ProcessName);
    foreach (var process in processes)

        /*если PID родительского процесса равен PID
        *текущего процесса*/

        if(Process.GetCurrentProcess().Id ==
            GetParentProcessId(process.Id))

            //запускаем метод

            func(process);
}
```

Теперь мы можем перейти к обработке событий элементов управления. В обработчиках нажатия на добавленные нами кнопки мы будем использовать методы манипулирования процессами, доступные как компонентные методы класса `Process`:

- `Kill` – принудительно завершает процесс;
- `CloseMainWindow` – закрывает главное окно процесса, чем приводит к завершению процесса. Это можно увидеть по тому, что после нажатия на кнопку `CloseWindow` нашего приложения произойдёт отработка события `Exited` класса `Process`;
- `Refresh` – очищает всю информацию ассоциированную с процессом, которая была занесена в кэш процесса.

```
/*обработчик события нажатия на кнопку Start
*основного диалога*/
private void StartButton_Click(object sender,
                                EventArgs e)
{
    RunProcess(AvailableAssemblies.SelectedItem.
               ToString());
}
void Kill(Process proc)
{
    proc.Kill();
}
/*обработчик события нажатия на кнопку Stop основного
*диалога*/
private void StopButton_Click(object sender,
                               EventArgs e)
{
```

```

        ExecuteOnProcessesByName(StartedAssemblies.
            SelectedItem.ToString(), Kill);
        StartedAssemblies.Items.Remove
            (StartedAssemblies.SelectedItem);
    }
    void CloseMainWindow(Process proc)
    {
        proc.CloseMainWindow();
    }
    /*обработчик события нажатия на кнопку Close
    *основного диалога*/
    private void CloseWindowButton_Click(object sender,
                                           EventArgs e)
    {
        ExecuteOnProcessesByName(StartedAssemblies.
            SelectedItem.ToString(),
            CloseMainWindow);
        StartedAssemblies.Items.Remove
            (StartedAssemblies.SelectedItem);
    }
    void Refresh(Process proc)
    {
        proc.Refresh();
    }
    /*обработчик события нажатия на кнопку Refresh
    *основного диалога*/
    private void Refresh_Click(object sender, EventArgs
    e)
    {
        ExecuteOnProcessesByName(StartedAssemblies.
            SelectedItem.ToString(), Refresh);
    }
    /*обработчик события изменения индекса выделенного
    *элемента в списке доступных приложений*/
    private void AvailableAssemblies_
        SelectedIndexChanged(object sender, EventArgs e)
    {

```

```

        if (AvailableAssemblies.SelectedItems.Count == 0)

            StartButton.Enabled = false;
        else

            StartButton.Enabled = true;
    }
    /*обработчик события изменения индекса выделенного
    *элемента в списке запущенных приложений*/
    private void StartedAssemblies_
    SelectedIndexChanged(object sender, EventArgs e)
    {
        if (StartedAssemblies.SelectedItems.Count == 0)
        {

            StopButton.Enabled = false;

            CloseButton.Enabled = false;

            CloseWindowButton.Enabled = false;
        }
        else
        {

            StopButton.Enabled = true;

            CloseButton.Enabled = true;

            CloseWindowButton.Enabled = true;
        }
    }
    /*обработчик события закрытия основного окна
    *приложения*/
    private void MainWindow_FormClosing(object sender,
    FormClosingEventArgs e)
    {

```



```
foreach (var proc in Processes)

    proc.Kill();
}
/*обработчик события нажатия на кнопку "Run Calc"*/
private void RunCalc_Click(object sender, EventArgs e)
{
    RunProcess("calc.exe");
}
```

После того, как описаны обработчики событий, можно переходить к запуску приложения.

## 5. Домен приложения

Обычно, операционная система и среда исполнения предоставляют некоторую форму изоляции приложений друг от друга. Это разделение необходимо для того, чтобы существовала некоторая степень уверенности, что код, исполняющийся в рамках одного приложения, не может повлиять на код, исполняющийся в другом, не связанном с ним, приложении. Поскольку отсутствие изоляции может привести к тому, что, вследствие сбоя в одном приложении, перестают работать сразу несколько приложений, или, что ещё хуже, происходит сбой системы в целом. Для изоляции исполняемого кода приложений операционная система Windows использует концепцию процессов, которая была описана выше.

Домены приложений используются для изоляции в области безопасности, надёжности, контроля версий, а так же для закрытия загруженных сборок в целях освобождения используемой ими памяти. Домены приложений, обычно, создаются "хостами среды исполнения" (от англ. runtime hosts), которые отвечают за настройку среды исполнения до того, как приложение будет запущено. Задача "runtime host"-а состоит в том, чтобы загрузить среду исполнения в процесс, создать домены приложений внутри процесса, а так же загрузить пользовательский код в домены приложений.

Для организации межпроцессного взаимодействия необходимо использовать некоторый объект-посред-

ник или прокси (от англ. proxy – "посредник"), который определял бы уровень разыменования.

Прежде, чем управляемый код будет выполнен, он должен пройти процесс проверки (верификации | verification process), исключая те случаи, когда администратор разрешает пропустить этот процесс. Проверка состоит в том, чтобы удостовериться, что код может получать доступ к недействительным адресам памяти или осуществлять какие-либо другие действия, которые потенциально могут привести к неправильной работе приложения. Код, который прошёл процесс верификации, называют type-safe кодом.

Возможность единой среды исполнения проверять, является ли код type-safe, позволяет предоставлять уровень изоляции, разграничивая процессы между собой с меньшим расходом производительности.

Домены приложений представляют собой наиболее безопасную и гибкую конструкцию, которую единая среда исполнения может предложить для осуществления изоляции между процессами. В рамках одного процесса можно запустить несколько доменов приложения с таким же уровнем изоляции, который будет существовать в нескольких отдельных процессах, но без необходимости реализовывать дополнительные действия по реализации вызовов или переключения между процессами.

Возможность запускать несколько приложений в рамках одного процесса увеличивает масштабируемость и гибкость решения (под решением, в данном контексте, понимается программный комплекс, реша-

ющий определённую задачу и являющийся целью процесса разработки).

В свою очередь, изоляция приложения важна для его безопасности.

Изоляция, предоставляемая доменами приложений, имеет следующие положительные стороны:

- ошибки исполнения одного приложения не могут повлиять на другое приложение, поскольку type-safe код не приводит к ошибкам обращения к памяти;
- отдельные приложения, запущенные в рамках одного процесса, могут быть остановлены без необходимости остановки процесса в целом (однако, необходимо помнить, что вы не можете выгрузить отдельные сборки или типы данных – только полностью весь домен приложения);
- код, исполняемый в рамках одного приложения, не может напрямую получить доступ к ресурсам или осуществить вызов кода другого приложения. Единая среда исполнения реализует такую изоляцию посредством блокирования прямых обращений между объектами находящимися в разных доменах приложений;
- поведение кода ограничено областью видимости того домена приложения, в рамках которого оно запущено. Домен приложения предоставляет настройки, такие как версия приложения, место нахождения сборок, к которым домен приложения получает доступ, а так же где находятся сборки, которые были загружены в текущий домен приложения;

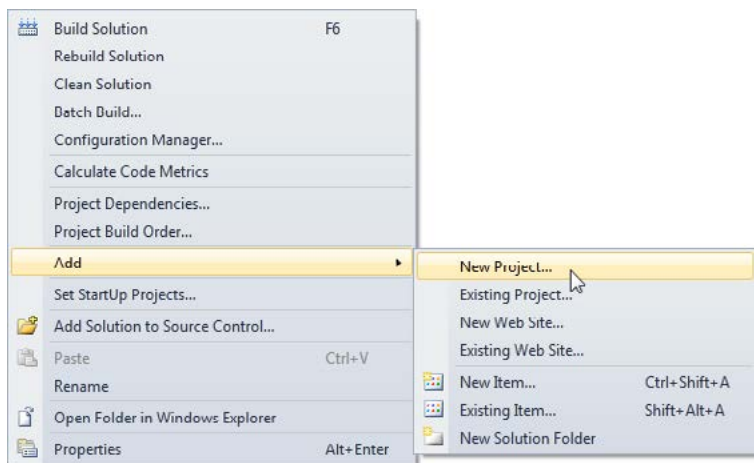
- политики безопасности, которые распространяются на код, могут контролироваться доменом приложения, в котором этот код выполняется.

Остановимся немного на вопросе взаимодействия домена приложения со сборками, которые он загружает.

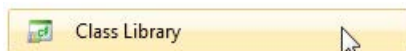
Прежде, чем выполнить некоторый код, необходимо загрузить сборку, которая его содержит в некоторый домен приложения. Как правило, приложение загружает сразу несколько сборок.

Ниже будет приведён пример приложения, которое динамически создаёт домен приложения, загружает в него dll-библиотеку и выполняет метод из этой библиотеки. После того, как искомый метод будет выполнен и библиотека будет более не нужна, приложение отгружает, созданный им, домен приложения, а соответственно и все ресурсы, с ним связанные.

Для начала необходимо создать проект по шаблону C# Console Application (консольное приложение C#). Мы назовём его AppDomainDynamicUnload. После этого в проект необходимо добавить dll-библиотеку, которую мы назовём SampleLibrary. Для этого необходимо вызвать контекстное меню на решении в окне Solution Explorer, выбрать в нём пункт Add и в появившемся выпадающем меню выбрать пункт Add Project, как показано на приведённом ниже изображении.



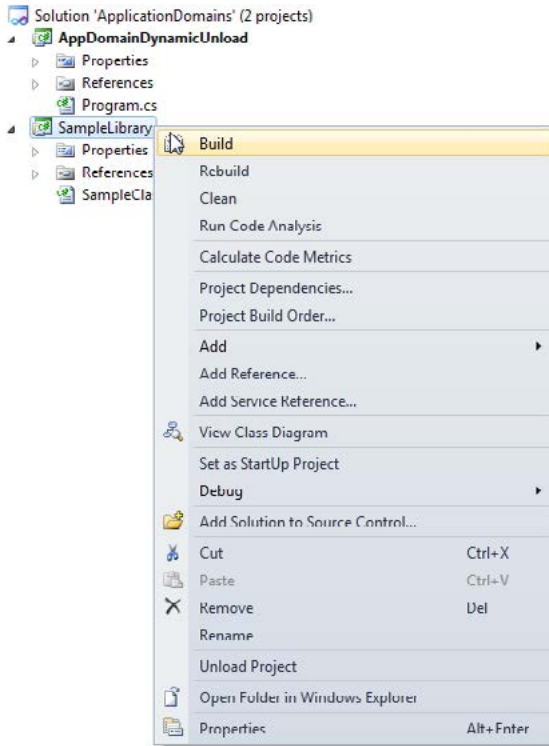
В появившемся окне необходимо выбрать тип шаблона Class Library.



В проекте сгенерируется файл Class1 – его необходимо переименовать в SampleClass. В этом файле необходимо объявить статический тип данных SampleClass, в котором нужно объявить общедоступный статический метод DoSome, без аргументов и возвращаемого значения.

```
public static class SampleClass
{
    public static void DoSome()
    {
        Console.WriteLine("Doing something!");
    }
}
```

Чтобы откомпилировать отдельный проект решения необходимо в окне solution Explorer для проекта вызвать контекстное меню и выбрать в нём пункт Build, как показано на приведённом ниже рисунке.



Получившийся в результате компиляции файл необходимо скопировать в директорию `./bin/debug` проекта `AppDomainDynamicUnload`.

В методе `Main` проекта `AppDomainDynamicUnload` необходимо реализовать динамическое создание домена приложения с загрузкой сборки `SampleLibrary.dll` в созданный нами домен и последующей выгрузкой до-

мена из приложения. Мы будем действовать по следующему алгоритму:

1. создаём объект `AppDomain` (описатель домена приложения) с произвольным именем;
2. загружаем в домен приложения сборку `SampleLibrary.dll` и сохраняем описатель в переменной типа `Assembly`. Для динамической загрузки сборки мы будем использовать метод `Load` класса `AppDomain`;
3. получаем описатель модуля, содержащего описание необходимого типа данных, инкапсулирующего искомый метод, который мы намереваемся вызвать. Для оперирования с модулем мы используем класс `Module`;
4. из полученного модуля, при помощи вызова метода `GetType` получаем тип данных, который сохраняем в переменную тип `Type`;
5. из полученного типа данных, посредством вызова метода `GetMethod` получаем объект класса `MethodInfo`, описывающего искомый нами метод;
6. вызываем метод `DoSome` класса `SampleClass` сборки `SampleLibrary` посредством вызова метода `Invoke` класса `MethodInfo`;
7. выгружаем домен приложения посредством вызова метода `Unload` класса `AppDomain`.

Далее приведён код метода `Main` приложения `AppDomainDynamicUnload`:



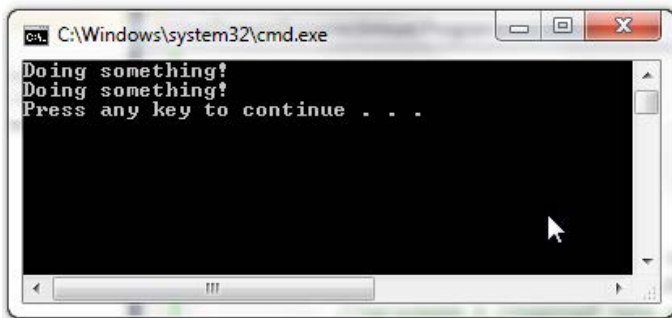
```

//создаём домен приложения с произвольным именем
AppDomain Domain = AppDomain.CreateDomain
                        ("Demo Domain");
//загружаем в созданный нами домен приложения заранее
//подготовленную dll библиотеку
Assembly asm = Domain.Load(AssemblyName.
                        GetAssemblyName("SampleLibrary.dll"));
//получаем модуль, из которого будем осуществлять
//вызов
Module module = asm.GetModule("SampleLibrary.dll");
//получаем тип данных, содержащий искомый метод
Type type = module.GetType("SampleLibrary.
                        SampleClass");
//получаем метод из типа данных
MethodInfo method = type.GetMethod("DoSome");
//осуществляем вызов метода
method.Invoke(null, null);
//одноточный вариант вызова того же метода через
//анонимные объекты
asm.GetModule("SampleLibrary.dll").
GetType("SampleLibrary.SampleClass").

        GetMethod("DoSome").Invoke(null, null);
//отгружаем домен приложения
AppDomain.Unload(Domain);

```

Результат работы приложения будет выглядеть подобным образом:



Способ, которым была загружена сборка, определяет, сможет ли JIT-скомпилированный код (Just-In-Time compiled code) быть доступен сразу нескольким доменам приложений в рамках процесса, и может ли сборка быть выгруженной из процесса:

- Если сборка загружена домен-нейтрально (domain-neutral), то все домены приложений, которые имеют один и тот же уровень доступа может получать доступ к одному и тому же JIT-скомпилированному коду, что уменьшает объём памяти, используемой приложением. Но, в таком случае, сборка не сможет быть выгружена из приложения, пока не будет завершён процесс.
- Если же сборка загружена не домен-нейтрально, она должна быть JIT-скомпилирована внутри домена приложения, который её загружает. В таком случае, сборка может быть выгружена посредством "отгрузки" (завершения) всех доменов приложения, которые её используют.

Перед тем, как загрузить среду исполнения в процесс, хост среды исполнения определяет загружать ли сборку как домен-нейтральную. Для управляемых приложений можно применить атрибут `LoaderOptimizationAttribute` к методу, который является точной входа в процесс. И указать соответствующее значение из перечисления `LoaderOptimization`. Для неуправляемых приложений, которые принимают (используют) единую среду исполнения можно указать соответствующий флаг при вызове метода `CorBindToRuntimeEx`.

Существует три варианта загрузки домен-нейтральных сборок:

- **SingleDomain** – ни одна сборка не загружается как домен-нейтральная, исключая Mscorlib, которая всегда загружается домен-нейтрально. Эта настройка называется SingleDomain (один домен), потому что используется в тех случаях, когда хост запускает только одно приложение внутри процесса.
- **MultiDomain** – загружает все сборки как домен-нейтральные. Используйте эту настройку, когда загружаете множество доменов приложений в рамках одного процесса, исполняющих один и тот же код.
- **MultiDomainHost** – загружает строго-именованные сборки как домен-нейтральные, если они и все их зависимости были указаны в глобальном кэше сборки. Остальные сборки загружаются и JIT-компилируются отдельно, каждая в том домене приложения, в котором она была загружена, и они могут быть выгружены из процесса. Используйте эту настройку, когда собираетесь запускать более одного приложения в рамках одного процесса, или когда имеет место набор сборок, который используется одновременно несколькими доменами приложений.

Применение атрибута `LoaderOptimizationAttribute` будет выглядеть следующим образом:

```
[LoaderOptimization(LoaderOptimization.  
                    SingleDomain)]  
static void Main(string[] args)  
{  
  
    //далее следует исполняемый код
```

JIT-скомпилированный код не может быть досту-

пен сразу нескольким сборкам, которые были загружены в `load-from` контексте, то есть с использованием метода `LoadFrom` класса `Assembly` или с использованием перегруженных вариантов метода `Load`, принимающего массив типа `byte`.

JIT-скомпилированный код сборки, содержащей точку входа в приложение, будет общедоступным только в том случае, если все зависимости названной сборки общедоступны.

Домен-нейтральная сборка может быть JIT-скомпилирована более чем один раз. Например, в том случае, если ограничения безопасности определённые для нескольких доменов приложений отличаются, то они не могут получать общий доступ к одному и тому же JIT-скомпилированному коду. Как бы то ни было, каждая копия JIT-скомпилированного кода будет общедоступна для доменов приложений с одними и теми же ограничениями безопасности.

При принятии решения о загрузке сборки как домен-нейтральной, необходимо учитывать следующие факторы:

- Доступ к статическим данным и методам медленнее для домен-нейтральных сборок, поскольку существует необходимость в изоляции сборки. Каждый домен приложения, который получает доступ к сборке должен иметь свою копию статических данных, во избежание ссылок на объекты в статических полях посредством меж-доменных обращений. Как результат, среда исполнения содержит определённую логику для перенаправления, осуществляющего вызов, объекта к необходимой копии статических данных или методу.

Эта избыточная логика замедляет вызов.

- Все зависимости сборки должны быть определены и загружена в момент загрузки домен-нейтральной сборки, поскольку зависимости, которые не могут быть загружены домен-нейтрально, не позволят загрузить сборку домен-нейтрально.

Домены приложений определяют границы изоляции процесса в области безопасности, контроля версий, а так же отгрузки управляемого кода. Потоки – это конструкции операционной системы, которые используются единой средой исполнения для запуска исполняемого кода. Во времени исполнения, весь управляемый код загружается в соответствующий домен приложения и запускается управляемый поток.

В рамках домена приложения может существовать не только одни, а любое количество потоков в любой момент времени, а так же каждый отдельный поток не ограничивается одним доменом приложения. В любой момент времени любой поток выполняется в рамках некоторого домена приложения. Ноль, один или более потоков могут быть выполняемы в любом домене приложения. Среда исполнения отслеживает – какие потоки исполняются в каких доменах приложений. Вы можете определить домен приложения, в котором исполняется поток посредством вызова метода `GetDomain` класса `Thread`.

Домены приложений создаются и программно управляются хостами среды исполнения. Класс `AppDomain` – это программный интерфейс позволяющий создавать и манипулировать доменами приложений в рамках приложения.

## 6. Использование доменов приложения

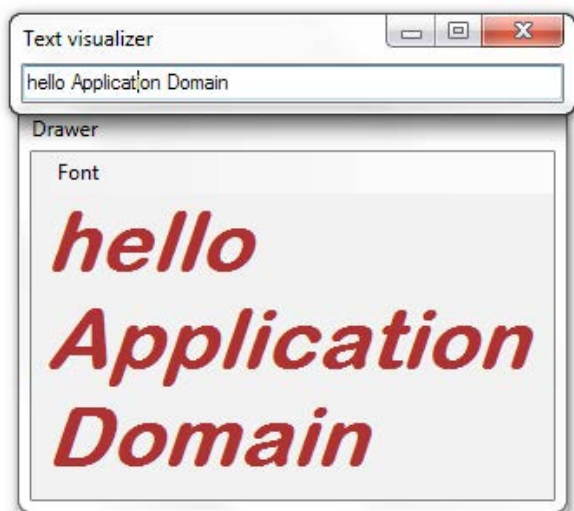
---

Использование доменов приложений мы рассмотрим на примере реализации оконного решения, которое на самом деле будет состоять из двух исполняемых (\*.exe) файлов, представляющих собой законченные приложения, но работающих в комплексе, и по отдельности не представляющих ценности.

Большинство сложных приложений состоят из множества мелких приложений. Иными словами имеют модульную структуру. Использование доменов приложений позволяет гибко моделировать подобную структуру с минимальным риском, поскольку при сохранении возможности взаимодействия ошибки, возникающие в одном приложении, не влияют на поведение и работоспособность остальных приложений, запущенных в рамках одного процесса.

Наше приложение будет состоять из двух окон: первое окно будет осуществлять приём от пользователя текстовой информации, а второе – осуществлять прорисовку введённого пользователем текста.

Следующее ниже изображение демонстрирует то, как будет выглядеть законченное приложение:



Для начала работы над проектом необходимо создать решение, которое будет называться UsingApp Domains. В него необходимо добавить три оконных проекта:

1. Собственно проект с названием UsingApp Domians;
2. Проект TextWindow;
3. Проект TextDrawer.

Окно проекта TextWindow необходимо настроить следующим образом:

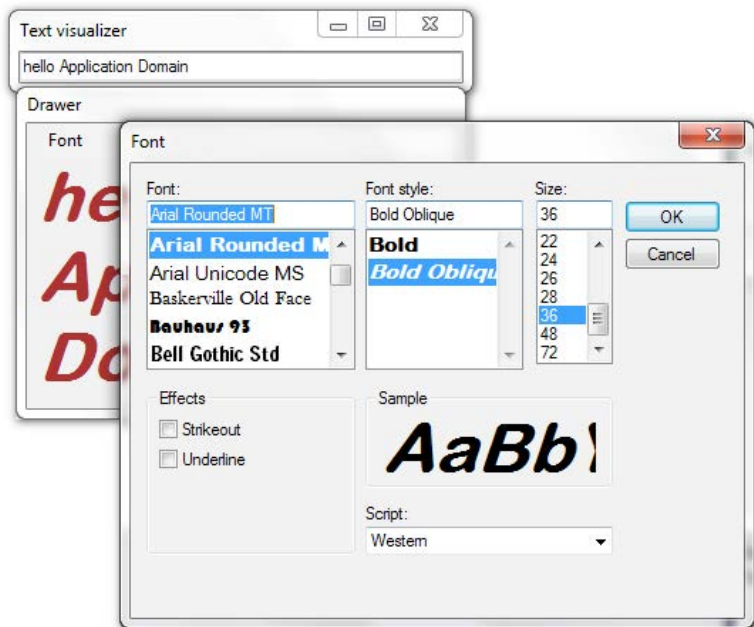
1. Свойство FormBorderStyle установить в значение FixedDialog;
2. Добавить в окно элемент управления TextBox и настроить габариты окна таким образом, чтобы они были оптимальными и не занимали лишне-

го экранного пространства. Должно получиться нечто подобное:



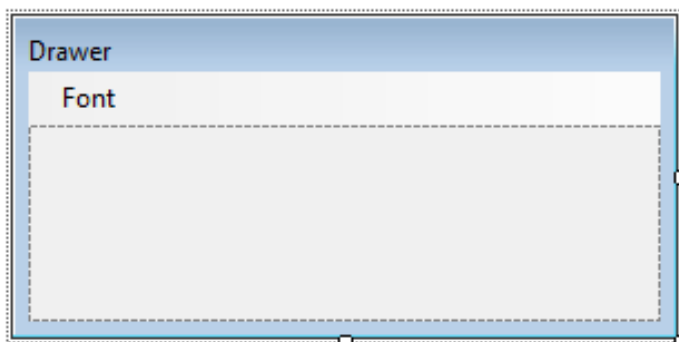
Окно проекта TextDrawer необходимо настроить в соответствии с указанными ниже требованиями:

1. Свойство `FormBorderStyle` установить в значение `SizableToolWindow`;
2. Свойство `ControlBox` установить в значение `false`;
3. Добавить на окно элемент управления `MenuStripe` и добавить пункт меню `Font`, который впоследствии будет вызывать диалог настройки шрифта, как показано на рисунке ниже;





4. Добавить на окно элемент управления Panel и свойство Dock этого элемента управления установить в значение Fill, чтобы он заполнил видимую клиентскую область окна. Должно получиться окно похожее на то, которое представлено на приведённом ниже рисунке.



Далее, из основного проекта (проект UsingApp Domains) необходимо удалить форму, которая сгенерировалась потому, что в шаблоне указано, что она должна быть. А, поскольку мы будем вручную запускать сборки, то это окно нам не нужно, нам нужна только точка входа в приложение.

Для начала необходимо описать структуру оконных классов нашего приложения.

В класс Form1 проекта TextDrawer необходимо добавить два поля:

```
string SourceText = "No text was added!";  
Font DrawingFont;
```

SourceText необходим для того, чтобы хранить текст, который будет прорисовываться на элементе управления Panel, а DrawingFont – предназначен для хранения настроек шрифта, выбранных пользователем.

В конструкторе необходимо проинициализировать переменную DrawingFont и добавить обработчики события Paint для окна и элемента управления Panel.

```
public Form1()
{
    InitializeComponent();
    DrawingFont = new Font("Arial", 45);
    panel1.Paint += Panel1_Paint;
    this.Paint += Form1_Paint;
}
```

В обработчике события Paint элемента управления Panel необходимо реализовать прорисовку текста с использованием выбранного шрифта.

```
private void Panel1_Paint(object sender,
PaintEventArgs e)
{
    if (SourceText.Length > 0)
    {
        /*создаём буферное изображение, основываясь на
        *размерах клиентской части элемента управления
        *Panel*/

        Image img = new Bitmap(panel1.ClientRectangle.
                                Width,
                                panel1.ClientRectangle.Height);
```

```

/*получаем графический контекст созданного нами
 *изображения*/

Graphics imgDC = Graphics.FromImage(img);

/*очищаем изображение используя цвет фона окна*/

imgDC.Clear(BackColor);

/*прорисовываем на элементе управления Panel
 *текст используя выбранный шрифт*/

imgDC.DrawString(SourceText, DrawingFont,
                  Brushes.Brown,
                  ClientRectangle,
                  new StringFormat(StringFormatFlags.
                                   NoFontFallback));

/*прорисовываем изображение на элементе
 *управления Panel*/

e.Graphics.DrawImage(img, 0, 0);
}
}

```

Обработчик события Paint окна будет вызывать аналогичный обработчик для элемента управления Panel.

```

void Form1 _ Paint(object sender, PaintEventArgs e)
{
    Panel1 _ Paint(panel1,
        new PaintEventArgs(panel1.CreateGraphics(),
        panel1.ClientRectangle));
}

```

На пункт меню Font, созданному нами ранее, необходимо добавить обработчик события Click, который будет выполнять следующую процедуру:

```
private void selectColorToolStripMenuItem_
    Click(object sender, EventArgs e)
{
    /*создаём объект стандартного диалога
    *FontDialog*/
    FontDialog dlg = new FontDialog();
    /*инициализируем объект шрифта для диалога*/
    dlg.Font = DrawingFont;
    /*открываем диалог модально*/
    if (dlg.ShowDialog() == DialogResult.OK)

        /*если была нажата кнопка OK, сохраняем
        *выбранные настройки*/

        DrawingFont = dlg.Font;
    /*инициируем перерисовку элемента управления
    *Panel*/
    Panell1_Paint(panell1,

        new PaintEventArgs(panell1.CreateGraphics(),
panell1.ClientRectangle));
}
```

Далее мы опишем два метода, которые в последствии будут нами вызываться из другого домена приложения: SetText – предназначенный для изменения текста и Move, предназначенный для управления положением окна.

```

public void SetText(string text)
{
    /*сохраняем новое значение переменной
    *SourceText*/
    SourceText = text;
    /*инициируем перерисовку окна*/
    Panel1 _ Paint(panel1,

        new PaintEventArgs(panel1.CreateGraphics(),
            panel1.ClientRectangle));
}

public void Move(Point newLocation, int width)
{
    /*устанавливаем новое значение позиции окна*/
    this.Location = newLocation;
    /*устанавливаем новое значение ширины окна*/
    this.Width = width;
}

```

В класс Form1 приложения TextWindow необходимо добавить два поля:

```

/*ссылка на модуль, из которого будем вызывать
*методы*/
Module DrawerModule { get; set; }
/*объект, от которого будем вызывать методы*/
object Drawer;

```

Так же необходимо переопределить конструктор, так, чтобы он принимал два параметра и инициализировал объявленные выше поля.

```
public Form1(Module drawer, object targetWindow)
{
    DrawerModule = drawer;
    Drawer = targetWindow;
    InitializeComponent();
}
```

Добавим обработчики событий изменения текста в текстовом поле и изменения положения окна.

```
private void textBox1_TextChanged(object sender,
                                   EventArgs e)
{
    /*вызываем метд SetText главного окна приложения
    *TextDrawer*/
    DrawerModule.GetType("TextDrawer.Form1").
        GetMethod("SetText").
        Invoke(Drawer, new object[]{textBox1.Text});
}

private void Form1_LocationChanged(object sender,
                                    EventArgs e)
{
    /*вызываем метд Move главного окна приложения
    *TextDrawer*/
    DrawerModule.GetType("TextDrawer.Form1").
        GetMethod("Move").
        Invoke(Drawer,
            new object[] {
                new Point(this.Location.X, this.Location.Y +
                           this.Height),
                this.Width });
}
```

В файле Program, проекта UsingAppDomains необходимо указать директивы using для двух пространств имён:

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

Пространство имён Reflection нам необходимо, поскольку мы будем использовать такие типы данных как MethodInfo и Assembly, а Threading – необходим для запуска первичных потоков наших приложений. Мы не будем останавливаться на теоретических сведениях о потоках, поскольку они будут описаны в шестой главе настоящего урока. Скажем только, что потоки это конструкции, которые операционная система использует для запуска исполняемого кода. Очевидно, чтобы процедуры выполнялись параллельно, необходимо, чтобы каждая была запущена в отдельном потоке. Из контекста будет понятно, что именно происходит, поэтому описание методов запуска и управления потоками мы оставим до соответствующей главы.

В классе Program необходимо объявить несколько статических переменных, как это показано ниже:

```
static AppDomain Drawer;           //будет хранить объект  
                                   //домена приложения TextDrawer  
  
static AppDomain TextWindow;       //будет хранить объект  
                                   //домена приложения TextWindow  
static Assembly DrawerAsm;         //будет хранить объект  
                                   //сборки TextDrawer.exe
```

```
static Assembly TextWindowAsm; //будет хранить объект
                                //сборки TextWindow.exe
static Form DrawerWindow;      //будет хранить объект
                                //окна TextDrawer
static Form TextWindowWnd      //будет хранить объект
                                //окна TextWindow
```

Методу, который является точкой входа в процесс, необходимо добавить атрибут `LoaderOptimization` в значении `MultiDomain`, для того, чтобы созданные нами домены имели доступ к исполняемому коду друг друга.

```
[STAThread]
[LoaderOptimization(LoaderOptimization.MultiDomain)]
static void Main()
{
```

Для того, чтобы визуализировать отгрузку одного из доменов приложения, то есть, тот факт, что домен приложения успешно отгружен, а значит и все сборки, которые он загрузил – тоже, мы создадим обработчик события `DomainUnload` класса `AppDomain`, который впоследствии добавим одному из созданных нами доменов приложений.

```
static void Drawer _ DomainUnload(object sender,
                                   EventArgs e)
{
    /*открываем MessageBox, в котором выводим имя
     *текущего домена приложения*/
    MessageBox.Show("Domain with name " +
```



```
(sender as AppDomain).FriendlyName +
    " has been succesfully unloaded!");
}
```

Наши домены должны выполняться параллельно, для того, чтобы реакцией на изменение текста в одном окне была мгновенная перерисовка текста в другом. Для обеспечения этой возможности мы каждый домен запустим в отдельном потоке. А значит для каждого из потоков нам необходимо описать потоковую процедуру.

```
static void RunDrawer()
{
    /*запускаем окно модально в текущем потоке*/
    DrawerWindow.ShowDialog();
    /*отгружаем домен приложения*/
    AppDomain.Unload(Drawer);
}
```

```
static void RunVisualizer()
{
    /*запускаем окно модально в текущем потоке*/
    TextWindowWnd.ShowDialog();
    /*завершаем работу приложения, следствием чего
    *станет закрытие потока*/
    Application.Exit();
}
```

После того, как сделаны подготовительные действия, мы можем перейти к описанию процесса создания доменов приложений, загрузки в домены необходимых сборок, а также запуска потоков нашего решения.

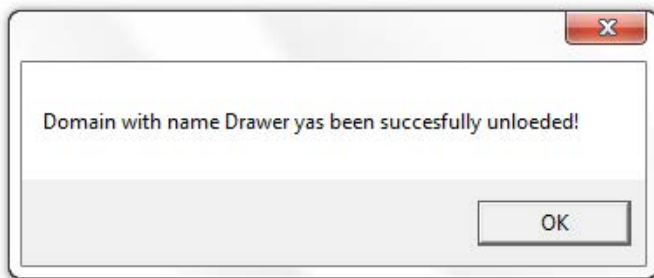
```

[STAThread]
[LoaderOptimization(LoaderOptimization.MultiDomain)]
static void Main()
{
    //включаем визуальные стили для приложения,
    //поскольку оно является оконным
    Application.EnableVisualStyles();
    /*создаём необходимые домены приложений с
     *дружественными именами и сохраняем ссылки
     *на них в соответствующие переменные*/
    Drawer = AppDomain.CreateDomain("Drawer");
    TextWindow = AppDomain.
        CreateDomain("TextWindow");
    /*загружаем сборки с оконными приложениями
     *в соответствующие домены приложений*/
    DrawerAsm = Drawer.Load(AssemblyName.
        GetAssemblyName("TextDrawer.exe"));
    TextWindowAsm = Drawer.Load(AssemblyName.
        GetAssemblyName("TextWindow.exe"));
    /*создаём объекты окон на основе оконных типов
     *данных из загруженных сборок*/
    DrawerWindow = Activator.
        CreateInstance(DrawerAsm.GetType("TextDrawer.
        Form1")) as Form;
    TextWindowWnd = Activator.CreateInstance(
        TextWindowAsm.GetType("TextWindow.Form1"),
        new object[]
        {
            DrawerAsm.
                GetModule("TextDrawer.exe"),
            DrawerWindow
        }) as Form;
    /*запускаем потоки*/
    (new Thread(new ThreadStart(RunVisualizer))).
    Start();
    (new Thread(new ThreadStart(RunDrawer))).Start();
    /*добавляем обработчик события DomainUnload*/
    Drawer.DomainUnload += new EventHandler(Drawer_
        DomainUnload);
}

```

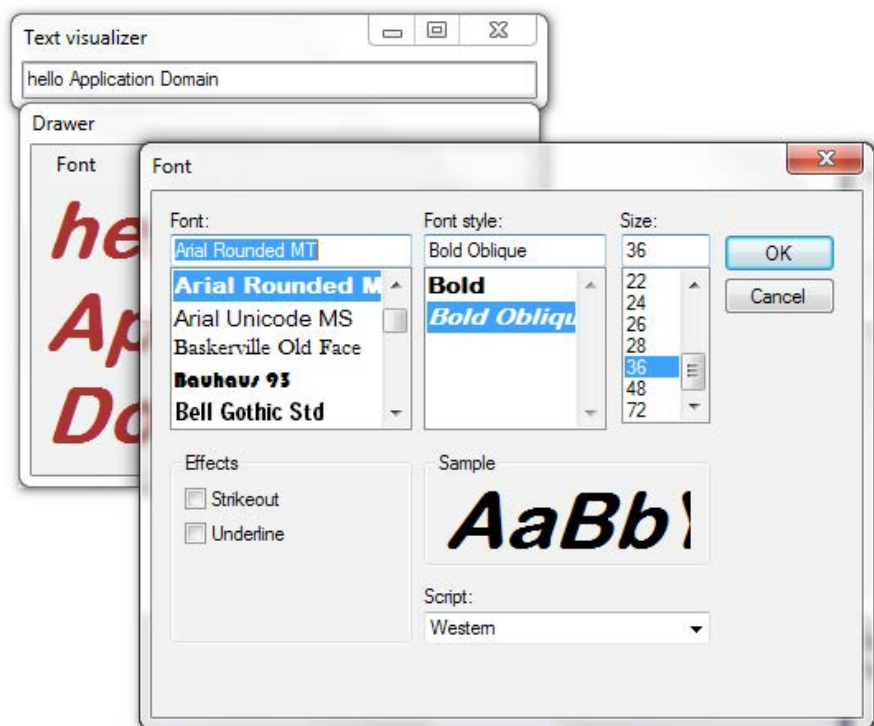
После запуска приложения, описанная выше процедура создаст два домена приложения и созданные нами проекты загрузит, каждый в отдельный домен приложения. После чего основываясь на оконных классах, описанных в загруженных сборках, будут созданы два оконных объекта, каждый из которых будет модально запущен в отдельном потоке.

При завершении потока, в котором запускается главное окно приложения `TextWindow`, иницируется завершение всего приложения, которое приводит к завершению процесса, а соответственно и всех, открытых в нём доменов приложений, о чём сигнализирует `MessageBox`, вызываемый в обработчике события `DomainUnload` домена приложения с именем "Drawer".



Так же, такой же `MessageBox` появляется как реакция на закрытие главного окна приложения `TextDrawer`, что свидетельствует о закрытии соответствующего домена приложения, а значит и об освобождении загруженных им сборок, хотя сам процесс и его остальные приложения всё ещё выполняются.

Работающее приложение будет выглядеть так, как показано на рисунке:



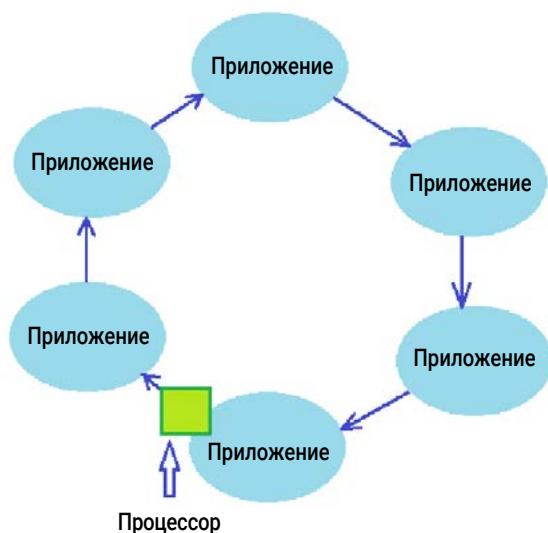
## 7. Многопоточность

---

*Многопоточность* – свойство платформы (например, операционной системы или приложения), состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся "параллельно". При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

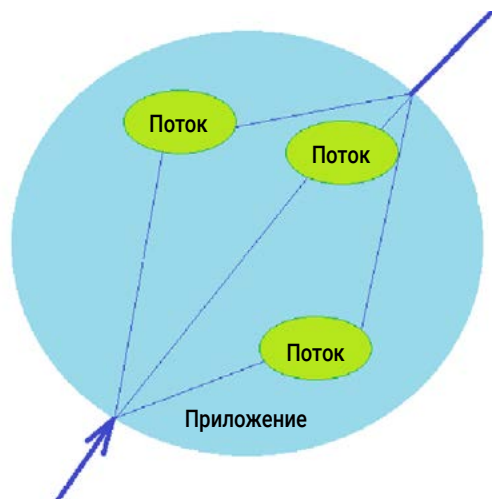
Физически многопоточности в системах под управлением Windows не существует, но существует логическая многопоточность, это достигается тем, что каждому процессу и потоку в нем, выделяется короткий отрезок времени для его работы, и каждый поток в приложении выполняется по очереди.

Для более простого понимания как работает система многозадачности, представим, что процессор это человек, который работает на нескольких работах утром он идет на первую работу, потом на вторую, после на следующую, и так пока не надо будет идти опять на первую работу. Процессор в своей работе напоминает такого человека, сначала он выполняет инструкции одной программы, потом второй, после третьей и так далее.



Процессор якобы движется по кругу, заходя к каждому приложению и выполняя его инструкции.

Внутри приложения потоки и процессор работают по тому же принципу, что и приложения:

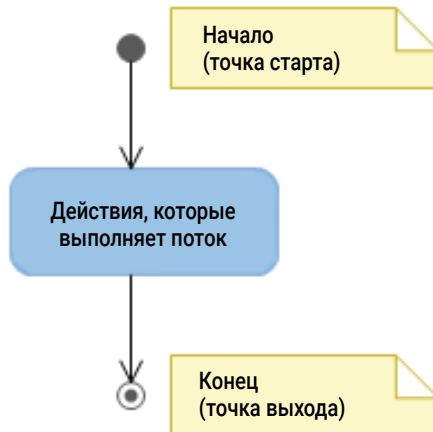


При "входе" в приложение процессор некоторое время выполняет инструкции одного из потоков, после чего переходит к следующему приложению или потоку.

Данное представление очень абстрактное, но дает понять, как физически устроена многопоточность.

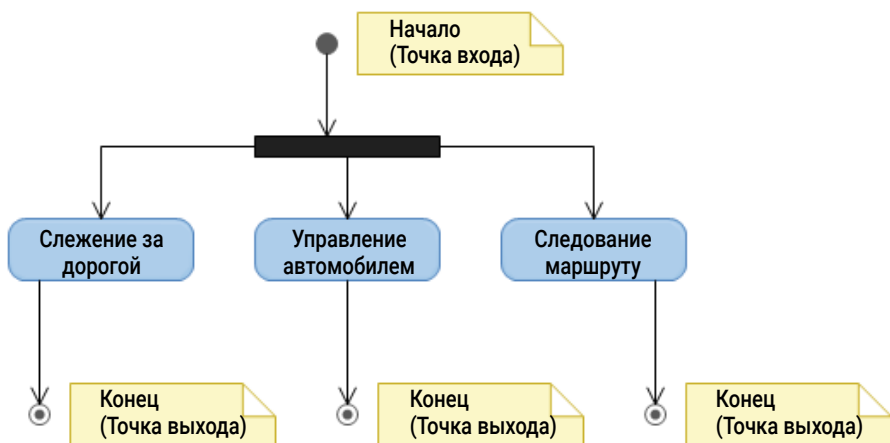
Реально, процессор сам выбирает количество времени, которое он будет слушать инструкции от потока (исходя из приоритета потока или приложения) и может прервать выполнение в любой момент. Поэтому логически потоки выполняются одновременно, и строить свои многопоточные приложения необходимо с учетом того что потоки работают одновременно и параллельно.

**Поток** – это не зависящая последовательность инструкций, в программе имеющая точку входа и точку выхода, другими словами это просто некоторый код, выполнение которого можно запустить параллельно с уже выполняемым. Такой инструмент позволяет более эффективно использовать вычислительную мощь процессора, и не тратить время на "простой" компьютера. Схематически поток отображается следующим образом:



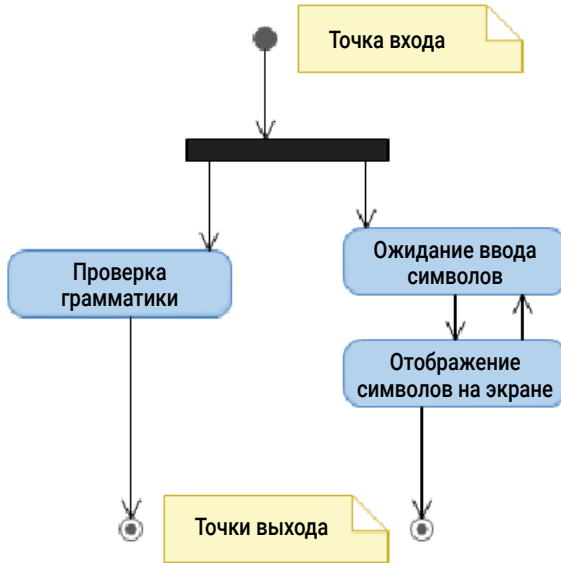
Многопоточность применяется в программах, которым одновременно нужно выполнять две или более задач, например: водитель, когда едет на автомобиле одновременно выполняет несколько задач, слежением за дорогой, управление транспортным средством, следование заранее спланированному маршруту. Гораздо эффективнее, когда водитель выполняет все эти действия параллельно, а не по очереди.

Схематическое представление действий водителя:



Более понятный для программистов пример – это текстовый редактор с фоновой проверкой грамматики (например, Microsoft Word), где в параллельно выполняются два потока: один ожидает от пользователя ввод символов, а второй циклично ищет в тексте ошибки. Схематическое представление этой программы:





Все потоки в .Net начинают свою работу с первого оператора метода который был вызван в новом потоке и заканчивают свою работу после того как выполнятся последний оператор этого метода. Например, основной поток программы начинает работу в начале метода `main` и завершает свою работу в конце функции `main`, для вторичных потоков программы нужно создавать другие функции и из основного потока запускать их в работу. Можно создавать несколько потоков, которые будут выполнять один и тот же код.

Поэтому всегда точка входа и точка выхода это начало и конец одного и того же метода.

При проектировании программы стоит учитывать тот факт, что потоки физически не выполняются одновременно, а выполняется часть каждого потока по очереди до завершения работы потока. Большим минусом

такой системы является то, что процессор тратит довольно много времени на переключение между потоками. Поэтому если перед вами стоит задача выполнить два действия максимально быстрее, лучше обойтись без многопоточности.

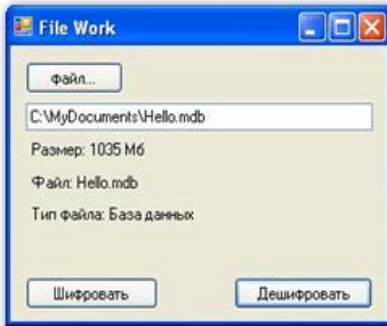
Но если в задаче имеется диалог с пользователем, например, программа ожидает от пользователя ввод символов из клавиатуры, или время от времени выводит пользователю информацию о ходе выполнения основной задачи. Такое приложение для большей эффективности следует сделать многопоточным, и диалог с пользователем поместить в отдельный поток, а в другой поток поместить выполнение основной задачи. При возможности стоит использовать такую технику программирования так как это приведет к большей эффективной затрате процессорного времени, потому что параллельно с инструкции в которых поток что-то "ждет" можно выполнить ресурсоемкую задачу.

Если речь идет других задачах, например работе с сетью, нужно следовать тем же принципам, так как при работе с сетью наверняка будут возникать задержки, потому можно использовать время процессора для задач в другом потоке.

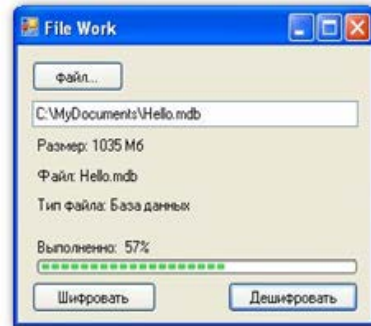
Рассмотрим пример опроса пользователей.

Имеется приложение которое выполняет чтение файла и шифрует/дешифрует его содержимое, программе для обработки 1ГБ данных надо 20 секунд.

Но программисты написали две версии приложения быструю и медленную:



Быстрая версия



Медленная версия

Быстрая версия выполняла свою работу в среднем 21 секунду а медленная версия выполняла свою работу за 35-40 секунд, но медленная версия выполняла задачу во вторичном потоке а в основном выводилась информация о ходе выполнения задачи.

Обе программы выданы пользователям и им был предоставлен выбор, какой пользоваться. Итог опроса пользователей показал, что подавляющее большинство (90%) пользуются медленной версией, а на вопрос "почему вы не пользуетесь программой, которая работает быстрее?" они, как правило, отвечали: "что программа часто зависает".

Все потому что: если задача выполняется в том же потоке где и прорисовывается окно, то на время выполнения задачи окно "зависает", а программу, которая не подает, признаков жизни пользователи пытаются закрыть, они вить не знают что она, сейчас просто работает.

А когда пользователь видит полосу загрузки, он может наблюдать, что программа работает.

## **Взаимодействие потоков**

Так как потоки "живут" в одной среде они могут взаимодействовать друг с другом, и за этим взаимодействием надо следить, ниже описаны основные проблемы, которые могут вам, встретиться при программировании в многопоточном режиме.

### **Голодание (starvation)**

Задержка времени от пробуждения потока до его вызова на процессор, в течение которой он находится в списке потоков, готовых к исполнению. Возникает по причине присутствия потоков с большими или равными приоритетами, которые исполняются все это время.

Негативный эффект заключается в том, что возникает задержка времени от пробуждения потока до исполнения им следующей важной операции, что задерживает исполнение этой операции, а следом за ней и работу многих других компонент.

Голодание создаёт узкое место в системе и не дает выжать из нее максимальную производительность, ограничиваемую только аппаратно обусловленными узкими местами.

Любое голодание вне 100 % загрузки процессора может быть устранено повышением приоритета голодающего потока, возможно – временным.

Как правило, для предотвращения голодания ОС автоматически вызывает на исполнение готовые к нему низкоприоритетные потоки даже при наличии высокоприоритетных, при условии, что поток не исполнялся в течение долгого времени (~10 секунд).

**Гонка (race condition)**

Недетерминированный порядок исполнения двух путей кода, работающих с одними и теми же данными и исполняемыми в двух различных нитях. Приводит к зависимости порядка – и правильности – исполнения от случайных факторов.

Устраняется добавлением необходимых блокировок и примитивов синхронизации. Обычно является легко устранимым дефектом (забытая блокировка).

***Синхронизация***

При программировании многопоточных приложений может возникать необходимость синхронизации потоков (упорядочиванию их выполнения) синхронизацию потоков необходимо выполнять для предотвращения проблем описанных выше, особенно когда несколько потоков работают с одними данными. Так же синхронизация необходима там где нужно просто упорядочить ход выполнения задач.

## 8. ПОТОКИ

Все инструменты для работы и управления потоками находятся в пространстве имен `System.Threading` для его подключения в Reference должна быть подключена библиотека `System`, которая добавляется автоматически при создании любого приложения работающего под управлением `.Net Framework`.

Рассмотрим несколько наиболее важных классов пространства имен **System.Threading**:

- *Thread* – класс создавая объект которого мы получаем возможность манипулировать понятием потока как привычным объектом. Например, останавливать поток и возобновлять его работу или задавать приоритет потока в программе.
- *Monitor, Mutex, Semaphore, Interlocked* – классы с которыми вы познакомитесь позже, они позволят строить логику синхронизации.
- *ThreadPool* – Статический класс предоставляющий доступ к пулу потоков. Пул потоков это коллекция потоков, которые, как правило, выполняются в фоновом режиме и призваны разгрузить основной поток от операций ожидания. Более детально мы с этим ознакомимся ниже.
- *Timer* – класс с помощью которого можно построить механизм вызова метода в заданные интервалы времени.

- *Делегаты и перечисления* из пространства имен `System.Threading` позволяют более удобно работать с потоками, их вы будете встречать ниже в связке с другими темами.

## Timer

Таймеры (Timer) окружают нас везде, начиная из светофоров на дорогах и заканчивая ядерными электростанциями. Поэтому у вас наверняка будут появляться идеи, где можно использовать таймеры.

Для создания и запуска таймера нужно выполнить 4 шага.

1. Описать метод, который будет выполняться по истечении определенного периода времени.

```
public static void TimerMethod(object a)
{
    Console.WriteLine("Hello in timer");
}
```

Аргумент `object a`, который вы видите в методе, будет принимать объект таймера, для того чтобы таймером можно было управлять из метода, который вызывается.

2. Создать объект делегата `TimerCallback` и связать с ним метод, который был описан выше.

```
TimerCallback callback = new
TimerCallback(TimerMethod);
```

3. Создать объект `Timer` и передать конструктор-делегат, который уже связан с методом, который будет вызываться с истечением таймера.

```
Timer timer = new Timer(callback);
```

4. Указать интервал таймера и запустить его, вызывая метод `Change` который имеет 3 перегрузки:

```
timer.Change(2000, 500);
```

В первом параметре метод принимает количество миллисекунд, которые он будет ждать перед запуском таймера, а второй параметр отвечает, с какой частотой будет вызываться метод обработчик (задается в миллисекундах).

После этих четырех действий таймер начнет работать. В данном случае через 2 секунды после вызова `Change` вызовется метод `TimerMethod`, и далее будет вызываться каждые пол секунды, работа метода `TimerMethod` будет выполняться в отдельном потоке от основного приложения.

Для завершения работы таймера необходимо просто вызвать метод `Dispose` из объекта таймера.

## Класс Thread

Класс `Thread` позволяет создавать потоки и управлять ими. Перед тем как создавать новый поток необходимо определиться будет ли поток принимать значения или будет вызываться метод без параметров.

Сначала мы рассмотрим вариант, где поток создается без параметров.

Так же как в работе с таймерами при создании пото-



ка необходимо описать метод, который будет работать в новом потоке:

```
static void Method()
{
    for (int i = 0; i < 1000; i++)
    {
        Console.WriteLine("\t\tHello in thread");
    }
}
```

И связать этот метод с специально созданным делегатом ThreadStart:

```
ThreadStart threadstart = new ThreadStart(Method);
```

Теперь можно приступить к созданию потока:

```
Thread thread = new Thread(threadstart);
```

И в завершение необходимо просто вызвать метод Start из объекта потока:

```
thread.Start();
```

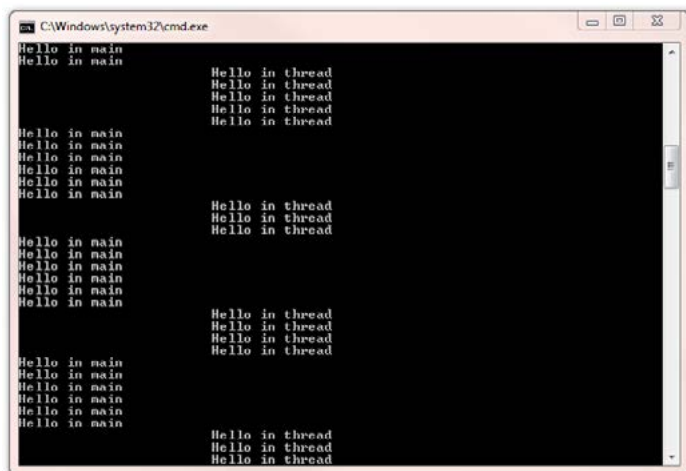
А для наглядности после создания и запуска нового потока, в основной поток можно добавить такую же логику, как и во вторичный – для более наглядного отображения их параллельной работы.

```

for (int i = 0; i < 100; i++)
{
    Console.WriteLine("Hello in main");
}

```

Результат работы программы будет примерно та-  
ким:



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output is interleaved, showing "Hello in main" and "Hello in thread" messages. The "main" thread outputs approximately 20 lines, and the "thread" thread outputs approximately 10 lines, demonstrating concurrent execution.

```

C:\Windows\system32\cmd.exe
Hello in main
Hello in main
Hello in main
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in main
Hello in main
Hello in main
Hello in main
Hello in main
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in main
Hello in main
Hello in main
Hello in main
Hello in main
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in main
Hello in main
Hello in main
Hello in main
Hello in main
Hello in thread
Hello in thread
Hello in thread
Hello in thread
Hello in thread

```

Как видите вывод из основного потока, где выводится "Hello in main" чередуется с выводом из созданного дополнительного потока, который выводит на экран строку "Hello in thread". Количество выведенных строк подряд из одного потока чередуется в зависимости от разных факторов системы, начиная от нагрузки системы заканчивая размером окна для вывода.

Создание потоков, которые принимают данные, несколько отличается от создания обычных потоков. Первое правило: поток может принимать входным параме-

тром только один объект типа `object`. Потому и метод, который должен будет вызываться в новом потоке тоже должен принимать один аргумент, типа `object`:

```
static void ThreadFunk(object a)
{
    string ID = (string)a;
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(ID);
    }
}
```

И далее связываем метод с делегатом, создаем объект потока и запускаем его в работу:

```
static void Main(string[] args)
{
    ParameterizedThreadStart threadstart =
        new ParameterizedThreadStart(ThreadFunk);
    Thread thread1 = new Thread(threadstart);
    thread1.Start((object)"One");
    Thread thread2 = new Thread(threadstart);
    thread2.Start((object)"\t\tTwo");
}
```

### Статические свойства класса `Thread`

У класса `Thread` имеется некоторое количество статических свойств и методов, рассмотрим некоторые из них.

```
Thread.Sleep(1000);
```

Статический метод `Sleep` из класса `Thread` позволяет остановить работу потока, который позволит приостановить на некоторое время работу потока, который получил эту инструкцию.

Многие думают, что `Sleep` останавливает работу только основного потока, но это не так, `Sleep` останавливает нить, которая выполняет этот код на время указанное в миллисекундах.

```
Thread ThisThread = Thread.CurrentThread;  
Console.WriteLine(ThisThread.GetHashCode());
```

Статическое свойство `CurrentThread` позволяет получить ссылку на текущий поток. Используя полученный объект `ThisThread` можно управлять потоком, который вызвал текущую операцию, другими словами поток будет управлять сам собой.

Если из потока вызвать метод `GetHashCode()` мы получим значение типа `int`, которое указывает, какой ID потока в этом приложении.

### Основные и вторичные потоки

Любой поток можно сделать вторичным (фоновым), основное отличие фонового потока от основного состоит в том, что если в приложении не останется ни одного работающего основного (не фонового) потока, то все фоновые потоки принудительно завершат свою работу.

Как правило, фоновые потоки используются для фоновых задач, например проверка грамматики в тексте, если пользователь просто закроет программу, которая

в фоновом режиме выполняет проверку грамматики, то программа закроется сразу, и не будет ждать пока, выполниться вся логика.

Для создания фонового потока необходимо создать обычный поток и задать ему свойство `isBackground = true` как в следующем примере кода:

```
ThreadStart ts = new ThreadStart(Method);  
    Thread t = new Thread(ts);  
    t.IsBackground = true;  
    t.Start();
```

По умолчанию свойство `isBackground` установлено `false`.

### Приостановка и возобновление работы потока

Порой возникают ситуации, когда есть необходимость остановить выполнение потока, например: пользователь желает временно приостановить процесс конвертации видео с одного формата в другой.

Для приостановки работы потока необходимо вызвать из-под объекта потока метод `Suspend`, а для возобновления работы потока необходимо вызвать метод `Resume`.

Ниже приведен код, который использует приостановку потока:

```
static void Main(string[] args)  
{  
    ThreadStart ts = new ThreadStart(Method);  
    Thread t = new Thread(ts);
```

```

t.Start(); // Запуск потока.
Console.WriteLine("Нажмите любую клавишу для
                  остановки");
Console.ReadKey();
t.Suspend(); // Приостановка потока.
Console.WriteLine("Поток остановлен!");
Console.WriteLine("Нажмите любую клавишу
                  для возобновления");
Console.ReadKey();
t.Resume(); // Возобновление работы.
}

static void Method()
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(i);
        Thread.Sleep(100);
    }
}

```

## Принудительное завершение работы потока

Потоки можно принудительно завершать, часто такая необходимость появляется, когда пользователь хочет отменить выполнение задачи.

Для принудительного завершения работы потока из его объекта, необходимо вызвать метод `Abort`.

```

ThreadStart ts = new ThreadStart(Method);
Thread t = new Thread(ts);
t.IsBackground = true;
t.Start();

```

При завершении работы потока в нем возникает исключение, это необходимо для того чтоб программисты могли обработать вызов `Abort`. Например: если в потоке выполнялась запись или чтение из файла и был вызван `Abort` для потока, то файл все равно необходимо закрыть, поэтому внутри метода, в котором работает поток необходимо писать блок `try`.

Пример оформления метода потока, который может быть принудительно завершен:

```
static void Method()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(100);

            Console.WriteLine(i.ToString());
        }
    }
    finally
    {
        Console.WriteLine("End Thread Work");
    }
}
```

В данном случае основная работа выполняется в конструкции `for`, но по завершении работы `for` или при принудительном завершении потока будет выполнен блок `finally`.

### Приоритеты потоков

Приоритет выполнения потоку назначается тогда, когда он выполняет более приоритетную задачу или задачу, которую нужно выполнить быстрее, чем остальные.

Приоритет потока назначается ему с помощью свойства потока `Priority`, которое является перечислением `ThreadPriority`.

Для программистов доступно 5 различных ступеней приоритета:

- Highest (Высокий);
- AboveNormal (Выше среднего);
- Normal (Средний);
- BelowNormal (Ниже среднего);
- Lowest (Низкий).

```
ParameterizedThreadStart ts =  
    new ParameterizedThreadStart(Method);  
Thread t1 = new Thread(ts);  
t1.Priority = ThreadPriority.Highest;  
t1.Start((object)"t1");
```

Для наглядного отображения того, что поток с большим приоритетом работает быстрее, можно написать следующее приложение:

```
static void Main(string[] args)
{
    ParameterizedThreadStart ts =
        new ParameterizedThreadStart(Method);
    Thread t1 = new Thread(ts);
    Thread t2 = new Thread(ts);
    t1.Priority = ThreadPriority.Highest;
    t2.Priority = ThreadPriority.Lowest;
    t2.Start((object)"\t\t\t2");
    t1.Start((object)"t1");
    Console.ReadKey();
}
```



```
static void Method(object str)
{
    string text = (string)str;
    for (int i = 0; i < 2000; i++)
    {
        Console.WriteLine("{0} #{1}",text,i.
ToString());
    }
}
```

Метод `Method` будет выполняться в новом потоке, вся работа, которую будет выполнять этот метод, заключается в прохождении цикла на 2000 итераций, при этом в каждой итерации находится команда, которая будет выводить на экран имя потока и номер итерации.

В основном потоке (метод `main`) создаются два потока, оба будут выполнять один и тот же метод, только первому потоку `t1` присваивается высокий приоритет, а потоку `t2` присваивается низкий приоритет, после чего оба потока запускаются на выполнение.

Результат работы программы такой:

```
C:\Windows\system32\cmd.exe
t2 #1845
t2 #1846
t1 #1967
t1 #1968
t1 #1969
t1 #1970
t1 #1971
t1 #1972
t1 #1973
t1 #1974
t1 #1975
t1 #1976
t1 #1977
t1 #1978
t1 #1979
t1 #1980
t1 #1981
t1 #1982
t1 #1983
t1 #1984
t1 #1985
t1 #1986
t1 #1987
t1 #1988
t1 #1989
t2 #1847
t2 #1848
t2 #1849
```

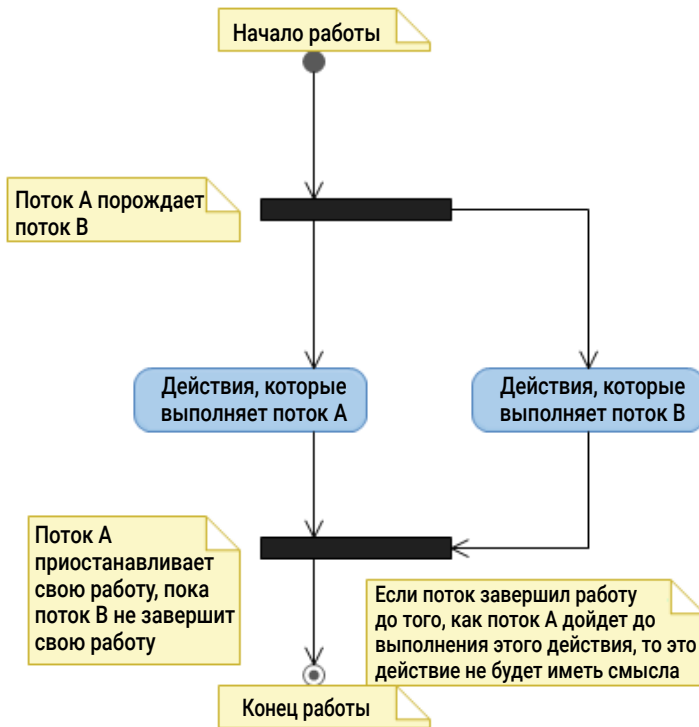
Как видите поток t1, выполнил уже 1989 итераций, в то время как поток t2 с низким приоритетом выполнил всего 1847 итераций. При увеличении объема выполняемых задач потоков разница между их выполнением будет только увеличиваться.

### Метод Join

У объекта потока есть метод Join, который позволяет выполнить следующие действия.

Поток, который вызывает метод Join из другого потока, будет ждать, пока завершит свое выполнение поток, из объекта которого вызвали Join. Другими словами, у нас есть поток А, который создает поток В, в котором выполняется некоторая задача. Теперь если поток А, вызовет из потока В метод Join то поток А будет ждать на месте вызова, пока не завершит свою работу поток В.

Схематическое представление:



Ниже представлен пример использования метода Join:

```

static void Main(string[] args)
{
    ThreadStart TS = new ThreadStart(Method);
    Thread T = new Thread(TS);
    Console.WriteLine("Сейчас будет запущен поток");

    T.Start();
    Thread.Sleep(200);
    Console.WriteLine("Ожидание завершения работы потока");
}
  
```

```
T.Join();  
    Console.WriteLine("Завершение работы  
                        программы");  
}  
static void Method()  
{  
    Console.WriteLine("Поток работает");  
    Thread.Sleep(2000);  
    Console.WriteLine("Поток завершил работу");  
}
```

Результат работы этой программы:



The screenshot shows a Windows command prompt window titled "cmd: C:\Windows\system32\cmd.exe". The output of the program is displayed as follows:

```
Сейчас будет запущен поток  
Поток работает  
Ожидание завершения работы потока  
Поток завершил работу  
Завершение работы программы
```

## 9. Практические примеры использования потоков

Используя потоки можно написать много интересных программ.

Ниже представлены исходные коды программы, которая выполняет шифрование и дешифрование файла.

```
// Создание делегата, который будет связан с методом
// шифрования или дешифрования
ParameterizedThreadStart Param = null;
while (true)
{
    // Меню, где пользователю предлагается выбрать
    // действие. Console.Clear();
    Console.WriteLine("1. Шифровать");
    Console.WriteLine("2. Дешифровать");
    ConsoleKeyInfo Select = Console.ReadKey(true);
    Console.Clear();
    if (ConsoleKey.D1 == Select.Key)
    {
        // Если пользователь выбрал шифрование,
        //мы связываем делегат с методом шифрования
        Param = new
        ParameterizedThreadStart(Encryption);
        Console.WriteLine("Введите путь к файлу, который
                           хотите зашифровать");
    }
    else if(ConsoleKey.D2 == Select.Key)
    {

```

```

        // Если пользователь выбрал дешифрование,
        // мы связываем делегат с методом дешифрования
Param = new ParameterizedThreadStart(Decryption);
Console.WriteLine("Введите путь к файлу, который
                    хотите расшифровать");
    }
    if(ConsoleKey.D1 == Select.Key || ConsoleKey.D2 ==
        Select.Key)
    {
        // Пользователь вводит путь к файлу,
        // с которым собирается работать
string FilePath = Console.ReadLine();
        // Создание потока, который будет шифровать
        // дешифровать
Thread thread = new Thread(Param);

        // Старт потока в параметр передается путь
        // к файлу
thread.Start((object)FilePath);
        Console.WriteLine("Нажмите символ, чтобы
                            выполнить действие");

do
    {
        // В цикле пользователю предлагаются выбрать
        // действие с потоком
        Console.WriteLine("[c] Отменить работу
                            потока");

        Console.WriteLine(

"[p] приостановить или возобновить работу
        потока");

        ConsoleKeyInfo Selects = Console.ReadKey(true);

        if (Selects.Key == ConsoleKey.C)
        {

            if (thread.ThreadState == ThreadState.Running)
            {

```

```

// Если поток выполнялся
// и пользователь выбрал завершение
thread.Abort();// Завершаем работу потока

Console.WriteLine("Поток остановлен");

}

}

else if (Selects.Key == ConsoleKey.P)
{
    if (thread.ThreadState == ThreadState.Running)
    {
        // Если поток выполнялся
        // и пользователь выбрал приостановление
        thread.Suspend();
        // Приостанавливаем поток
        Console.WriteLine("Поток
                               приостановлен");
    }
    else if (thread.ThreadState ==
              ThreadState.Suspended)
    {
        // если поток остановлен
        // и пользователь выбрал возобновление
        thread.Resume();
        // Возобновляем поток
        Console.WriteLine("Поток
                               восстановил работу");
    }
    Thread.Sleep(100);
}
}

```

```

        // Если поток приостановлен или работает,
        // то показать это меню пользователю еще
    } while (thread.ThreadState == ThreadState.
                Suspended ||
                thread.ThreadState == ThreadState.
                Running);

    Console.ReadKey(true);
    Console.Clear();
}
}

```

Методы Encryption и Decryption в примере не показаны, они здесь просто используются.

Многопоточность позволила этому приложению сделать так, чтобы пользователь мог управлять ходом шифрования (отмена, приостановка, возобновление).

Следующий пример, который мы рассмотрим, будет сервер многопользовательского сетевого приложения. Но для мобильности приложение будет только симулировать работу сети. Концепция касательно многопоточности останется такой же, как в реальных сетевых приложениях.

В приложениях такого типа, как правило, создается поток, который принимает подключения и после подключения создает и передает управление в новый поток, в котором будет, осуществляется работа именно с этим пользователем. После создания потока для пользователя поток, который "слушает" подключения, возвращается к приему подключений. Для каждого пользователя создается отдельный поток в приложении.



```

// Основной метод программы
static void Main(string[] args)
{
    // Создаем и запускаем "слушатель"
    ThreadStart Lis = new ThreadStart(LisenerClient);
    Thread LisenerThread = new Thread(Lis);
    LisenerThread.IsBackground = false;
    LisenerThread.Start();
}
// Метод, который будет выполняться в отдельном потоке
// и будет ждать подключений от пользователей
static void LisenerClient()
{
    int Counter = 0;
    while (true)
    {
        // Нажатием кнопки пользователь программы симулирует
        // сетевое подключения пользователя к серверу
        Console.WriteLine("Нажмите любую клавишу
                           для симуляции подключения");
        Console.ReadKey(true);
        ParameterizedThreadStart UserDel =
            new ParameterizedThreadStart
                (UserThreadFunk);
        // Создание объекта потока
        // (Для каждого клиента)
        Thread UserWorkThread = new
                                Thread(UserDel);

        // Запуск потока
        UserWorkThread.Start((object)Counter.
                               ToString());

        Counter++;
    }
}
// Метод будет выполняться в отдельном потоке,
// для каждого подключенного клиента
static void UserThreadFunk(object a)
{

```

```
string UserName = (string)a;
Console.WriteLine("пользователь\t# "+UserName+"
                  подключился");
while (true)
{
    // Ожидание команды пользователя
    switch (GetUserCommand())
    {
    case 0:
        Console.WriteLine("# {0} подписался на
                           новости" , UserName );
        break;
    case 1:
        Console.WriteLine("# {0} начал чат",
                           UserName);
        break;
    case 2:
        Console.WriteLine("# {0} купил продукцию
                           в магазине", UserName);
        break;
    case 3:
        Console.WriteLine("# {0} отправил письмо",
                           UserName);
        break;
    case 4:
        Console.WriteLine("# {0} отключился",
                           UserName);
        return; // Завершение потока
    }
}
```

Ниже представлено данное приложение в работе.

```

C:\Windows\system32\cmd.exe
пользователь # 7 начал чат
пользователь # 6 отправил письмо
пользователь # 2 отправил письмо
пользователь # 9 Отключился
пользователь # 8 подписался на новости
пользователь # 7 купила продукцию в магазине
пользователь # 6 отправил письмо
пользователь # 7 купила продукцию в магазине
пользователь # 8 отправил письмо
пользователь # 6 Отключился
пользователь # 7 купила продукцию в магазине
пользователь # 2 начал чай
пользователь # 8 купила продукцию в магазине
пользователь # 8 подписался на новости
пользователь # 8 Отключился
пользователь # 7 Отключился
пользователь # 2 купила продукцию в магазине
пользователь # 2 начал чай
пользователь # 2 купила продукцию в магазине
пользователь # 2 отправил письмо
пользователь # 2 купила продукцию в магазине
пользователь # 2 купила продукцию в магазине

```

Данную конструкцию сетевых приложений вы наверняка еще будете применять при написании многопользовательских приложений.

## 10. Домашнее задание

1. Создайте поток, который "принимает" в себя коллекцию элементов, и вызывает из каждого элемента коллекции метод `ToString()` и выводит результат работы метода на экран.
2. Создайте класс `Bank`, в котором будут следующие свойства: `int money`, `string name`, `int percent`. Постройте класс так, чтобы при изменении одного из свойств, класса создавался новый поток, который записывал данные о свойствах класса в текстовый файл на жестком диске. Класс должен инкапсулировать в себе всю логику многопоточности.
3. Реализуйте консольное игровое приложение "успел, не успел", где будет проверяться скорость реакции пользователя. Программа должна подать сигнал пользователю в виде текста, и пользователю должен будет нажать кнопку на клавиатуре, после нажатия пользователь должен увидеть, сколько миллисекунд ему потребовалось, чтобы нажать кнопку.
4. Реализуйте оконное приложение, которое в виде дерева выводит список всех запущенный в операционной системе процессов. Дерево должно отражать подчинённое отношение процессов по отношению друг к другу.

Корневые узлы дерева должны представлять собой процессы, у которых отсутствуют родительские процессы, а их дочерние узлы должны представлять их дочерние процессы. Предусмотрите в окне вывод ин-

формации о том процессе, который выделен в дереве.

Реализуйте приложение так, чтобы алгоритм получения информации о процессах выполнялся в дочернем потоке и не останавливал работу первичного потока приложения, в котором реализован интерфейс пользователя. Другими словами, дочерний поток должен будет выстраивать структуру процессов в элементе управления `TreeView`, параллельно с тем, как первичный поток будет выводить информацию о выбранном пользователем процессе.

5. Реализуйте приложение, которое позволит строить графики на основании таблиц данных. Например, статистику изменения температуры воздуха за месяц.

Приложение должно состоять из двух независимых сборок, каждая из которых должна иметь возможность запускаться как самостоятельное приложение и при этом не возвращать ошибки.

Создайте проект, который запустит эти приложения как связанные, в двух независимых доменах приложений. Первое оконное приложение должно принимать от пользователя информацию и инициировать прорисовку графика во втором приложении, которое также должно поддерживать возможность сохранения полученного графика как изображения на жёсткий диск.

В качестве принципа организации модульной структуры приложения, можно использовать пример, приведённый в главе 6 текущего урока, но не стоит сразу отрицать альтернативные решения.