

# Платформа Microsoft .NET и язык программирования C#



# Урок №12

## Использование унаследованного программного кода

## Содержание

1.	Почему необходимо использовать унаследованный программный код .....	4
2.	Пространство System.Runtime.InteropServices .....	6
3.	Взаимодействие с модулями Dll .....	7
4.	Класс DllImportAttribute .....	9
	Поле ExactSpelling .....	9
	Поле EntryPoint .....	11
	Поле CharSet .....	14
	Поле CallingConvention .....	15
	Поле SetLastError .....	17
5.	Примеры использования .....	19
6.	Создание DLL на языке C# .....	25

7. Global Assembly Cache.....	31
Что такое Global Assembly Cache? .....	31
Цели и задачи GAC .....	32
Строгие имена и версии DLL .....	33
Инсталляция сборок в GAC .....	37
8. Класс Assembly .....	39
Загрузка сборок .....	41
9. Механизмы безопасности в .Net Framework(Code Access Security) .....	48
10. Домашнее задание .....	54

# 1. Почему необходимо использовать унаследованный программный код

Унаследованный программный код — довольно широкое определение, под которым понимается некий, не всегда хороший, код «доставшийся в наследство» от другого разработчика или компании либо гарантировано качественные наработки целой технологии. Первый вариант мы здесь рассматривать не будем, поговорим о втором.

Решая поставленную задачу, вы можете быть уверены (почти со 100% вероятностью) в том, что где-то, кто-то и когда-то уже решал нечто подобное, поэтому чтобы не «изобретать велосипед» вы можете воспользоваться результатами предшественников, если конечно они не против. Скорее всего вы не найдете полного решения своей задачи, однако если вы получите хотя бы часть решения, то вам останется только адаптировать его под свою задачу. Мы не призываем вас списывать или красть чужую информацию, речь идет о различных модулях, как платных, так и выложенных в свободном доступе в сети Интернет. Обычно такие модули оформляются в виде динамически подключаемой библиотеки — *Dynamic Link Library* (DLL).

Ваше приложение, написанное на языке C#, может взаимодействовать с библиотеками, созданными с использованием как управляемых, так и неуправляемых языков программирования.

Если библиотека DLL написана на любом из языков семейства .NET, тогда вам останется только подключить эту библиотеку в своем приложении. Конечно же, вы можете оформить и собственный класс в виде DLL для повторного его использования в своих приложениях, как это сделать мы обсудим в шестом разделе. Таким образом, ваше приложение может взаимодействовать с так называемым управляемым кодом, который выполняется под управлением CLR, что помимо всего прочего гарантирует корректную работу с выделенной памятью.

Языки платформы .NET Framework содержат большое количество классов, структур и т.д., однако не все задачи можно решить доступными средствами. И хотя такие ситуации встречаются не часто, у вас существует возможность воспользоваться функциями, написанными на неуправляемых языках программирования, например использовать функции из библиотек WIN API на языке C++. Следует учитывать, что выполнение неуправляемого кода происходит вне среды CLR, непосредственно в процессоре, что может привести ко множеству проблем, в частности к утечке памяти.

Еще одной причиной использования неуправляемых функций в вашем приложении является невозможность получения исходного кода этих функций при помощи рефлексоров, то есть вы можете поместить в функцию, написанную на языке C++, блок кода, содержащий какие-то секретные данные, например аутентификацию пользователей.

## 2. Пространство System.Runtime.InteropServices

Пространство имен System.Runtime.InteropServices содержит большое количество классов, структур, перечислений и интерфейсов, которые обеспечивают взаимодействие между приложениями, написанными на .NET-языках программирования, и функциями управляемого API. Основные из них будут рассмотрены в последующих разделах.

### 3. Взаимодействие с модулями DLL

При использовании в .NET-приложении неуправляемого кода функций из библиотек DLL, необходимо выполнить ряд действий.

Во-первых, обязательным условием является объявление функций DLL в пределах класса и, хотя для этих целей можно использовать любой класс, рекомендуется создать отдельный класс и поместить в него используемые функции. Размещение неуправляемых функций в отдельном управляемом классе позволяет инкапсулировать весь процесс взаимодействия с ними в одном месте, что упростит дальнейшее использование этих функций.

В полученном классе для каждой используемой функции DLL необходимо создать прототип, представленный в виде статического метода с ключевым словом `extern`, определение которого может содержать дополнительные параметры. Для того чтобы это осуществить, необходимо указать имя функции и название библиотеки, где находится реализация соответствующей функции. Выполняя эти действия, вы как бы упаковываете неуправляемые функции DLL в управляемые методы вашего класса, после этого вы сможете вызывать их наравне со всеми остальными методами вашего приложения. При вызове такого метода выполняется определенная последовательность действий: определение библиотеки DLL для неуправляемой

функции, загрузка этой библиотеки в память, выполнение необходимого преобразования данных (маршалинг) и вызов самой неуправляемой функции. Ключевую роль при создании прототипа неуправляемой функции DLL играет класс `DllImportAttribute`.

# 4. Класс DllImportAttribute

Класс `DllImportAttribute` представляет, применяемый только к методам, атрибут, который обеспечивает возможность подключения неуправляемой DLL, содержащей требуемую функцию, например:

```
[DllImport("User32.dll")]
static extern int MessageBox(IntPtr h,
                            string m,
                            string c,
                            int type);
```

В данном примере чтобы обеспечить вызов неуправляемой функции `MessageBox()` необходимо подключить библиотеку `User32.dll`. В ряде случаев использования класса `DllImportAttribute` достаточно указать только название подключаемой библиотеки, однако, у данного атрибута существует набор полей, установка значений которых позволяет корректировать выполнение неуправляемой функции. Рассмотрим наиболее часто используемые поля этого класса.

## Поле ExactSpelling

Данное поле определяет, должно ли изменяться имя функции в зависимости от кодировки. Дело в том, что функции WIN API позволяют работать со строками в двух форматах кодирования Unicode и ANSI и различаются последней буквой `W` и `A` соответственно, например `MessageBoxW()` и `MessageBoxA()`.

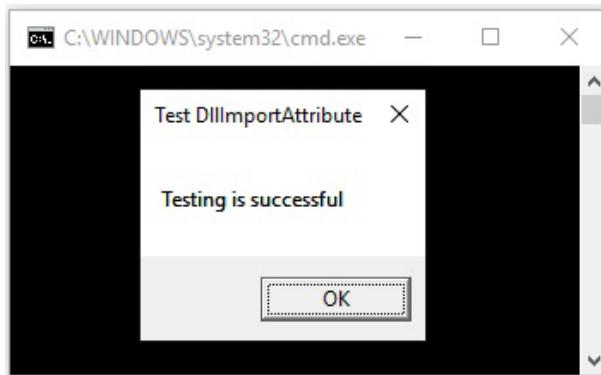
Если значение поля `ExactSpelling` установить в `true`, то поиск функции будет осуществляться по точно совпадающему имени. В языке C# значение по умолчанию этого поля равно `false`, и в этом случае вначале осуществляется поиск точного псевдонима функции, затем, если псевдоним не найден, будет осуществлен поиск по добавленному имени в зависимости от параметра поля `CharSet` (описано ниже). Если найти псевдоним функции не удается, генерируется исключительная ситуация `EntryPointNotFoundException`.

```
using System;
using System.Runtime.InteropServices;

namespace SimpleProject
{
    public class DllImportExample
    {
        [DllImport("User32.dll", ExactSpelling = true)]
        public static extern int MessageBoxA(IntPtr hWnd,
                                            string text, string caption, uint type);
    }

    class Program
    {
        static void Main(string[] args)
        {
            DllImportExample.MessageBoxA(IntPtr.Zero,
                                         "Testing is successful",
                                         "Test DllImportAttribute", 0);

        }
    }
}
```



**Рисунок 4.1.** Пример использования поля ExactSpelling

При выполнении этой программы вызывается функция MessageBoxA(), использующая формат ANSI. IntPtr.Zero применяется для передачи указателя, который определен в соответствии с размером указателя базовой платформы, что значительно упрощает преобразование в 64-разрядные платформы.

## Поле EntryPoint

Это поле позволяет явным образом указать имя вызываемой неуправляемой функции, вследствие чего вы можете использовать произвольное имя для вашего статического метода. Если название управляемого метода совпадает с именем неуправляемой функции, тогда в использовании поля EntryPoint нет необходимости.

В следующем примере неуправляемая функция GetVersionEx(), которая позволяет получить информацию об установленной операционной системе, будет вызываться при помощи управляемого статического метода GetVersion() (Рисунок 4.2).

```
using System.Runtime.InteropServices;
using static System.Console;

namespace SimpleProject
{

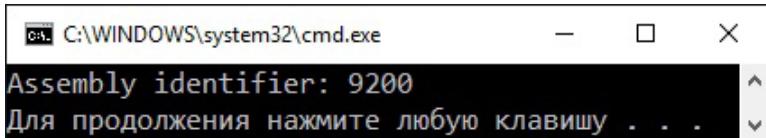
    public unsafe struct OsVersionInfo
    {
        public uint osVersionInfoSize;
        public uint majorVersion;
        public uint minorVersion;
        public uint buildNumber;
        public uint platformId;
        public fixed byte servicePackVersion[128];
    }

    public class DllImportExample
    {
        [DllImport("Kernel32.dll",
                   EntryPoint = "GetVersionEx")]
        public static extern bool GetVersion(ref
                                              OsVersionInfo versionInfo);
    }

    class Program
    {
        static void Main(string[] args)
        {
            OsVersionInfo versionInfo =
                new OsVersionInfo();
            versionInfo.osVersionInfoSize =
                (uint)Marshal.SizeOf(versionInfo);
            bool result = DllImportExample.
                GetVersion(ref versionInfo);
            if (result)
            {

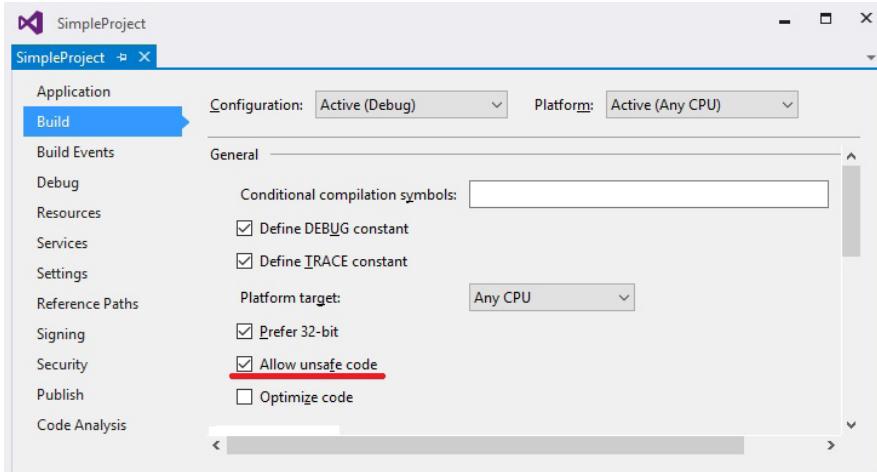
```

```
        WriteLine($"Assembly identifier:  
                  {versionInfo.buildNumber}");  
    }  
}  
}  
}  
}
```



**Рисунок 4.2.** Пример использования поля EntryPoint

В предыдущем примере использовался небезопасный код, то есть код, который обычно используется в языке C++. Участки небезопасного кода в коде на языке C# помечаются при помощи ключевого слова `unsafe`. Для того чтобы разрешить использование небезопасного кода



**Рисунок 4.3.** Разрешение использования небезопасного кода

в приложении на языке C# необходимо открыть окно свойств проекта, для чего в пункте Project главного меню Visual Studio 2015 необходимо выбрать пункт, оканчивающийся на Properties.... В открывшемся окне выбрать вкладку Build, а на ней установить флажок напротив пункта Allow unsafe code (Рисунок 4.3).

Ключевое слово `fixed` запрещает перемещение определенного объекта (в нашем случае массива) сборщиком мусора при выполнении дефрагментации памяти, для того чтобы в указателе на этот объект всегда находился конкретный адрес памяти.

В предыдущем примере также используется класс `Marshal`, предоставляющий набор методов, которые могут использоваться для работы с различными типами неуправляемой памяти.

Всегда следует помнить, что прибегать к использованию небезопасного кода в управляемых приложениях необходимо только в самых крайних случаях, когда без этого нельзя обойтись, например при работе с указателями в унаследованном коде на языке C++ или необходимости в прямом доступе к памяти для повышения производительности.

### Поле CharSet

При помощи этого поля можно задать кодировку строк при преобразовании неуправляемого кода в управляемый, для этого необходимо указать одно из значений перечисления `CharSet`. В языке C# значение поля CharSet по умолчанию равно `CharSet.Ansi` (строки представляются в виде однобайтовых ANSI-символов), при указании

значения `CharSet.Unicode` строки будут представлены в виде двухбайтовых символов Юникода.

```
using System;
using System.Runtime.InteropServices;

namespace SimpleProject
{
    public class DllImportExample
    {
        [DllImport("User32.dll", CharSet = CharSet.Auto)]
        public static extern int MessageBox(IntPtr hWnd,
                                           string text, string caption, uint type);
    }
    class Program
    {
        static void Main(string[] args)
        {
            DllImportExample.MessageBox(IntPtr.Zero,
                                         "Testing is successful",
                                         "Test DllImportAttribute", 0);
        }
    }
}
```

Результат выполнения этой программы соответствует рисунку 4.1, но в данном случае среда выполнения будет автоматически использовать соответствующую функцию (`MessageBoxA ()` или `MessageBoxW ()`) в соответствии с целевой платформой, благодаря установке значения `CharSet.Auto` поля `CharSet`.

## Поле CallingConvention

При помощи поля `CallingConvention` можно указать необходимые действия, связанные с техническими

особенностями вызова неуправляемого кода. Параметры этого поля позволяют оптимизировать вызов неуправляемой функции, регулируя способы ее вызова, передачи параметров и возврата результатов ее работы.

В качестве значения данного поля присваивается одно из значения перечисления  `CallingConvention`, по умолчанию  `CallingConvention.Winapi`. В следующем примере при вызове функции `printf()` полю  `CallingConvention` задается значение  `CallingConvention.Cdecl`, так как стек должен очищаться вызывающим объектом (Рисунок 4.4).

```
using System.Runtime.InteropServices;

namespace SimpleProject
{
    public class DllImportExample
    {
        [DllImport("msvcrt.dll", CharSet = CharSet.Ansi,
                   CallingConvention =
                   CallingConvention.Cdecl)]
        public static extern int printf(string format,
                                       int i, double d);

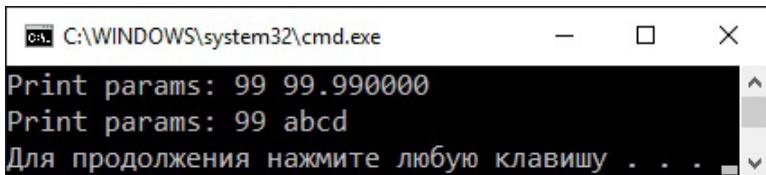
        [DllImport("msvcrt.dll", CharSet = CharSet.Ansi,
                   CallingConvention =
                   CallingConvention.Cdecl)]
        public static extern int printf(string format,
                                       int i, string s);
    }

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

```

        DllImportExample.printf("Print params:
            %i %f\n", 99, 99.99);
        DllImportExample.printf("Print params:
            %i %s\n", 99, "abcd");
    }
}
}

```



**Рисунок 4.4.** Пример использования поля CallingConvention

## Поле SetLastError

Этому полю задается значение типа `bool`, которое предоставляет возможность вызова функции `SetLastError()` WIN API в случае возникновения ошибки при выполнении управляемого метода, в языке C# значение по умолчанию `false`.

Следующий пример демонстрирует возможность получения имени текущего пользователя компьютера при помощи функции `GetUserName()` из библиотеки `Advapi32.dll` (Рисунок 4.5).

```

using System.Runtime.InteropServices;
using System.Text;
using static System.Console;

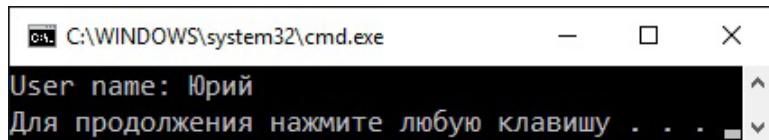
namespace SimpleProject
{

```

```
public class DllImportExample
{
    [DllImport("Advapi32.dll", SetLastError = true)]
    public static extern bool
        GetUserName(StringBuilder builder, ref int len);
}

class Program
{
    static void Main(string[] args)
    {
        int len = 20;
        StringBuilder builder = new StringBuilder(len);

        DllImportExample.GetUserName(builder, ref len);
        WriteLine($"User name: {builder}");
    }
}
```



**Рисунок 4.5.** Пример использования поля SetLastError

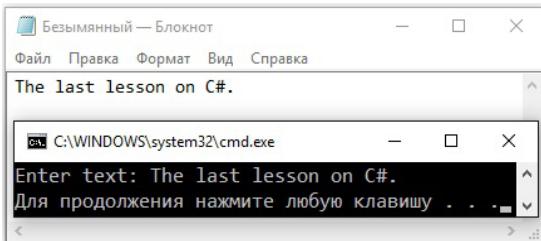
# 5. Примеры использования

В качестве примера мы приведем использование функции SetForegroundWindow() из библиотеки User32.dll, которая активирует окно, заданное через дескриптор, и переводит это окно в приоритетный режим (Рисунок 5.1).

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Windows.Forms;
using static System.Console;

namespace SimpleProject
{
    public class DllImportExample
    {
        [DllImport("User32.dll")]
        public static extern int
            SetForegroundWindow(IntPtr point);
    }
    class Program
    {
        static void Main(string[] args)
        {
            string processName = "notepad.exe", text;
            Write("Enter text: ");
            text = ReadLine();
            Process p = Process.Start(processName);
            p.WaitForInputIdle();
            IntPtr h = p.MainWindowHandle;
```

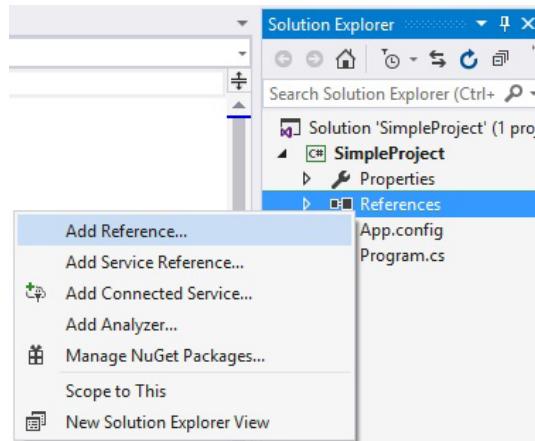
```
        DllImportExample.SetForegroundWindow(h);
        SendKeys.SendWait(text);
    }
}
}
```



**Рисунок 5.1.** Использование функции  
SetForegroundWindow()

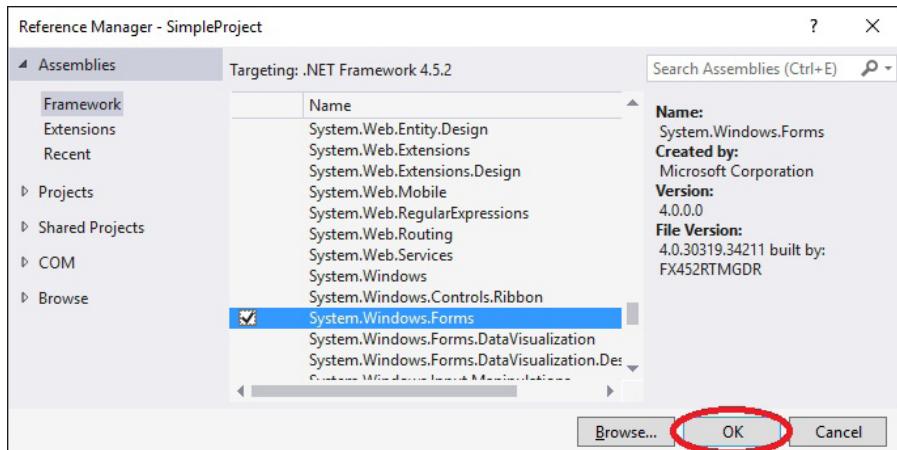
В приведенном примере в начале вызывается метод `Start()` класса `Process` для запуска указанного процесса (в нашем случае открываем стандартную программу Блокнот). Переводим этот процесс в состояние ожидания, вызывая метод `WaitForInputIdle()`, после чего при помощи свойства `MainWindowHandle` получаем дескриптор окна запущенного процесса и переводим это окно в приоритетный режим (метод `SetForegroundWindow()`). И, наконец, при помощи метода `SendWait()` класса `SendKeys` отправляем сообщение о нажатых клавишиах.

Класс `SendKeys` находится в пространстве имен `System.Windows.Forms`, которое необходимо подключить к текущему приложению, для чего надо нажать правой клавишей мыши на разделе `References` в окне `Solution Explorer` и выбрать пункт `Add Reference...` контекстного меню (Рисунок 5.2).



**Рисунок 5.2.** Подключение пространства имен к текущему проекту (начало)

В открывшемся окне, при выбранном пункте Framework, необходимо найти и отметить пространство имен System.Windows.Forms после чего нажать кнопку OK (Рисунок 5.3).



**Рисунок 5.3.** Подключение пространства имен к текущему проекту (продолжение)

В следующем примере мы используем собственную библиотеку SimpleCalc.dll, написанную на языке C++, исходный код которой приведен ниже.

```
__declspec(dllexport)
int add(int a, int b)
{
    return a + b;
}

__declspec(dllexport)
int sub(int a, int b)
{
    return a - b;
}

__declspec(dllexport)
int mult(int a, int b)
{
    return a * b;
}

__declspec(dllexport)
int div(int a, int b)
{
    if (b == 0)
    {
        throw;
    }
    return a / b;
}
```

Как вы видите, в этой библиотеке содержатся функции, позволяющие выполнять простые арифметические операции с двумя целочисленными значениями.

```
using System;
using System.Runtime.InteropServices;
using static System.Console;

namespace SimpleProject
{
    public class DllImportExample
    {
        [DllImport("SimpleCalc.dll")]
        public static extern int add(int a, int b);

        [DllImport("SimpleCalc.dll")]
        public static extern int sub(int a, int b);

        [DllImport("SimpleCalc.dll")]
        public static extern int mult(int a, int b);

        [DllImport("SimpleCalc.dll")]
        public static extern int div(int a, int b);
    }

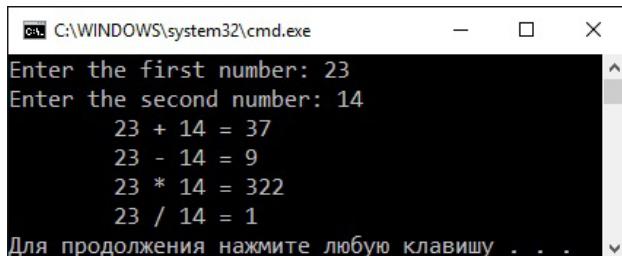
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Write("Enter the first number: ");
                int number1 = int.Parse(ReadLine());

                Write("Enter the second number: ");
                int number2 = int.Parse(ReadLine());

                WriteLine($"\\t{number1} + {number2} = {DllImportExample.add(number1, number2)}");
                WriteLine($"\\t{number1} - {number2} = {DllImportExample.sub(number1, number2)}");
            }
        }
    }
}
```

```
        WriteLine($"\\t{number1} * {number2} =\n        {DllImportExample.mult(number1,\n        number2)}");\n        WriteLine($"\\t{number1} / {number2} =\n        {DllImportExample.div(number1,\n        number2)}");\n    }\n    catch (Exception ex)\n    {\n        WriteLine(ex.Message);\n    }\n}\n}\n}
```

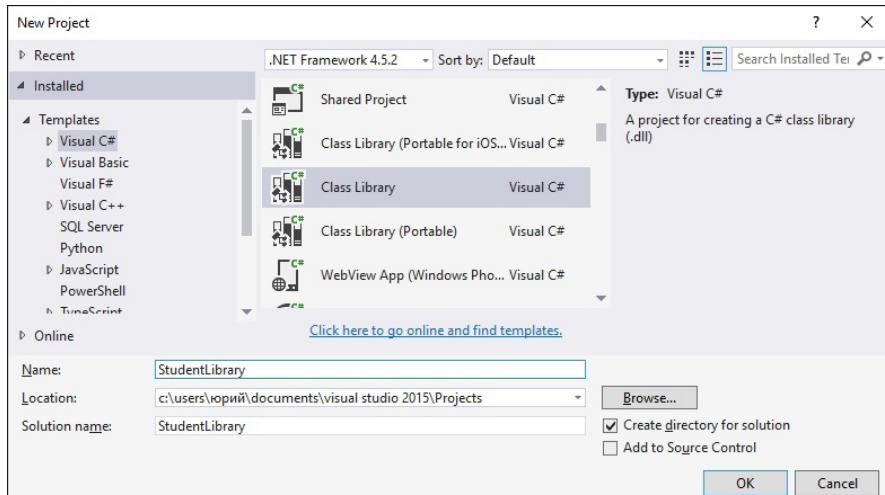
Возможный результат работы программы приведен на рисунке 5.4.



**Рисунок 5.4.** Использование функций из библиотеки SimpleCalc.dll

# 6. Создание DLL на языке C#

Для создания DLL на языке C# необходимо в Visual Studio 2015 создать новый проект типа Class Library, который назовем, например StudentLibrary (Рисунок 6.1).



**Рисунок 6.1.** Создание проекта типа Class Library

В созданной библиотеке по умолчанию находится класс **Class1**, в окне Solution Explorer мы переименуем его в **Student** и напишем несложную функциональность.

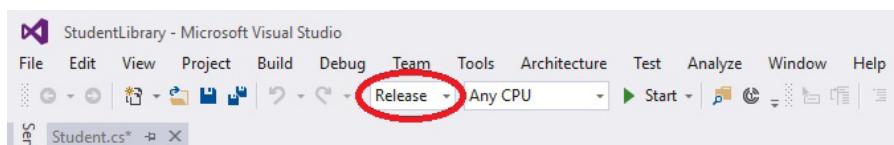
```
using System;

namespace StudentLibrary
{
    public class Student
    {
```

```
public string FirstName { get; set; }
public string LastName { get; set; }
public DateTime DateOfBirth { get; set; }

public override string ToString()
{
    return $"Surname: {LastName},
           Name: {FirstName},
           Date of Birth: {DateOfBirth.
ToLongDateString()}";
}
}
```

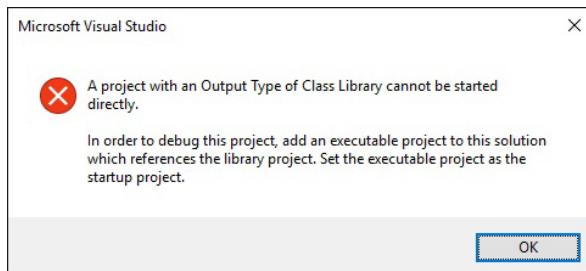
Перед компиляцией DLL изменим значение Debug в выпадающем списке Solution Configurations на панели инструментов на Release. Это рекомендуется делать для всех сборок, которые будут устанавливаться на целевой компьютер, так как в Release версии кода в отличие от Debug версии отсутствует отладочная информация (Рисунок 6.2).



**Рисунок 6.2.** Значение Release в выпадающем списке Solution Configurations

После этого мы скомпилируем нашу библиотеку, выбрав пункт Build Solution меню Build, то есть компиляция без запуска. Как вы думаете, почему нет смысла запускать на выполнение полученную библиотеку?

Если вы не смогли быстро ответить на этот вопрос, тогда можете осуществить запуск библиотеки на выполнение обычным способом с отладкой (клавиша F5) или без отладки проекта (сочетание клавиш Ctrl+F5). Библиотека скомпилируется, но вы увидите окно ошибки (Рисунок 6.3).

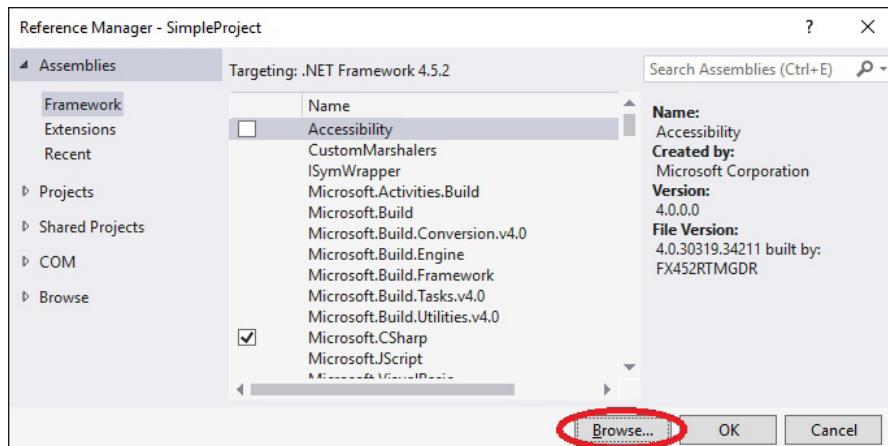


**Рисунок 6.3.** Сообщение об ошибке  
при компиляции библиотеки

В сообщении об ошибке говорится, о том, что для запуска этого проекта необходимо наличие исполняемого проекта. А чем характеризуется исполняемый модуль? Правильно, наличием метода `Main()`, который отсутствует в любой библиотеке.

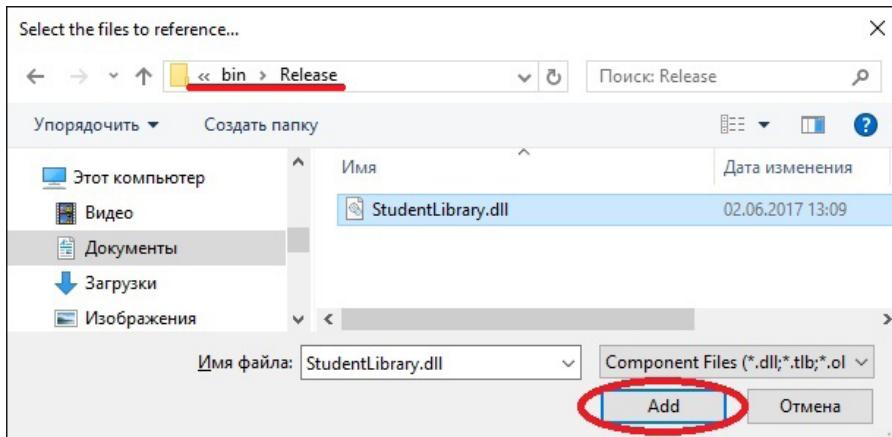
Полученная библиотека DLL будет находиться в папке `bin\Release` нашего проекта, теперь осталось привести пример ее использования в .NET-приложении. Для этого создадим консольное приложение, в котором будем использовать класс `Student` из созданной ранее библиотеки `StudentLibrary`. Для того чтобы приложение «видело» этот класс, необходимо подключить библиотеку `StudentLibrary`, для этого вначале надо выполнить действия, которые описывались в пятом разделе этого урока (Рисунок 5.2). Затем в открывшемся окне `Reference Manager` необходимо нажать кнопку `Browse...` (Рисунок 6.4).

## Урок №12



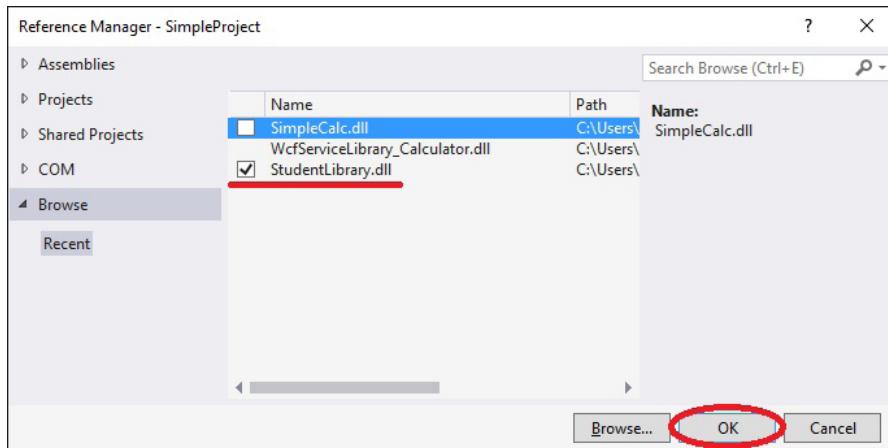
**Рисунок 6.4.** Подключение DLL к проекту (начало)

В открывшемся окне Select the files to reference... нужно выбрать путь к требуемой DLL (в нашем случае StudentLibrary), после чего нажать кнопку Add (Рисунок 6.5).



**Рисунок 6.5.** Подключение DLL  
к проекту (продолжение)

В списке библиотек окна Reference Manager следует убедиться, что требуемая библиотека отмечена, и закрыть это окно, нажав кнопку OK (Рисунок 6.6).



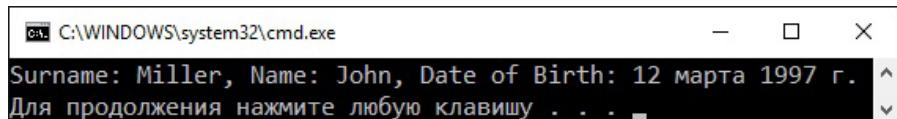
**Рисунок 6.6.** Подключение DLL  
к проекту (окончание)

После этого мы можем спокойно использовать класс **Student** в нашем приложении, подключив библиотеку **StudentLibrary** при помощи ключевого слова **using**, результат представлен на рисунке 6.7.

```
using StudentLibrary;
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
Student student = new Student
{
    FirstName = "John",
    LastName = "Miller",
    DateOfBirth = new DateTime(1997, 3, 12)
};
WriteLine(student);
}
}
```



**Рисунок 6.7.** Пример использования DLL на языке C#

# 7. Global Assembly Cache

Прежде всего, скажем несколько слов о размещении вашего готового приложения на целевом компьютере. Для этого нет необходимости в создании вами программ-инсталляторов (хотя и это тоже возможно), достаточно просто скопировать в папку на целевом компьютере файл с расширением .exe из папки Release (в крайнем случае, Debug) вашего проекта и все дополнительные библиотеки DLL. Обязательным условием работы вашей программы является наличие на целевом компьютере требуемой версии .NET Framework.

## Что такое Global Assembly Cache?

Если DLL находится в одном каталоге с программным кодом, то такая библиотека называется **частной сборкой** и именно программа отвечает за использование нужной версии этой библиотеки. Удаление частной сборки скажется на работе только приложения, которое ее использует.

Существуют также библиотеки, которые используются для работы множества приложений, например если вы развернете узел References в Solution Explorer своего проекта, то увидите ряд библиотек, которые не поставляются явным образом вместе с вашей программой, так как за их наличие на целевом компьютере отвечает BCL. Такие библиотеки DLL называются **разделяемыми сборками**.

Как вы уже поняли, разделяемые сборки не размещаются в одной папке с программным кодом, для этих целей

предназначен специальный каталог, который называется глобальный кэш сборок — *Global Assembly Cache* (GAC).

## Цели и задачи GAC

Глобальный кэш сборок расположен в подкаталоге assembly по следующему пути C:\WINDOWS\Microsoft.NET\assembly и предназначен для хранения библиотек DLL, которые используются при работе нескольких приложений.

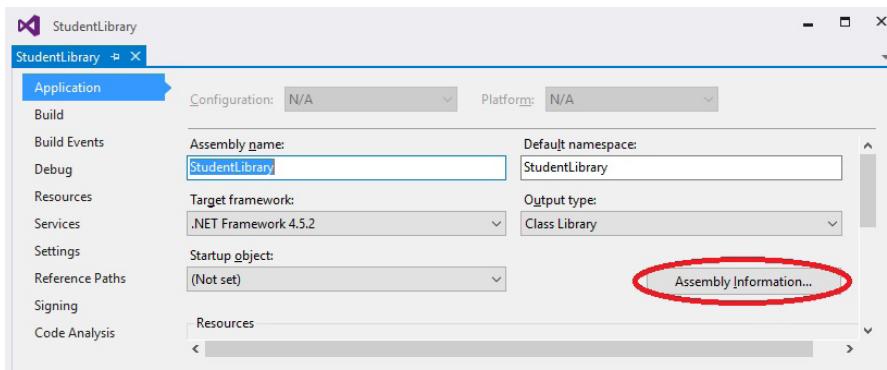
При помощи GAC решается проблема DLL Hell, которая связана с использованием библиотек DLL в операционной системе Windows. Данная проблема возникает при использовании несколькими приложениями различных версий одной и той же библиотеки DLL, основная трудность заключается в получении приложением необходимой DLL. В библиотеках DLL нет встроенного механизма обратной совместимости, и даже незначительные изменения в DLL делают ее внутреннюю структуру настолько отличной от предыдущих версий, что попытка их использования обычно приводит к сбою приложения. При наличии на компьютере версии DLL которая отличается от версии, используемой при создании программы, операционная система не может определить правильную версию DLL и «превращается в чертенка и идет в ад».

Частные сборки избегают этой проблемы, потому что версия DLL, которая использовалась для сборки приложения, включена в нее, поэтому даже если более новая версия существует в другом месте компьютера, это не влияет на работу данного приложения.

## Строгие имена и версии DLL

«Различные версии библиотеки можно разместить в любом каталоге» — скажете вы, однако для размещения DLL в GAC необходимо наличие у этой библиотеки версии и строгого имени, которые обеспечивают ее уникальность в глобальном кэше сборок. При компиляции проекта, использующего такую библиотеку, в его метаданных сохраняется имя библиотеки, ее версия и строгое имя, а CLR при вызове приложения теперь будет искать именно эту библиотеку.

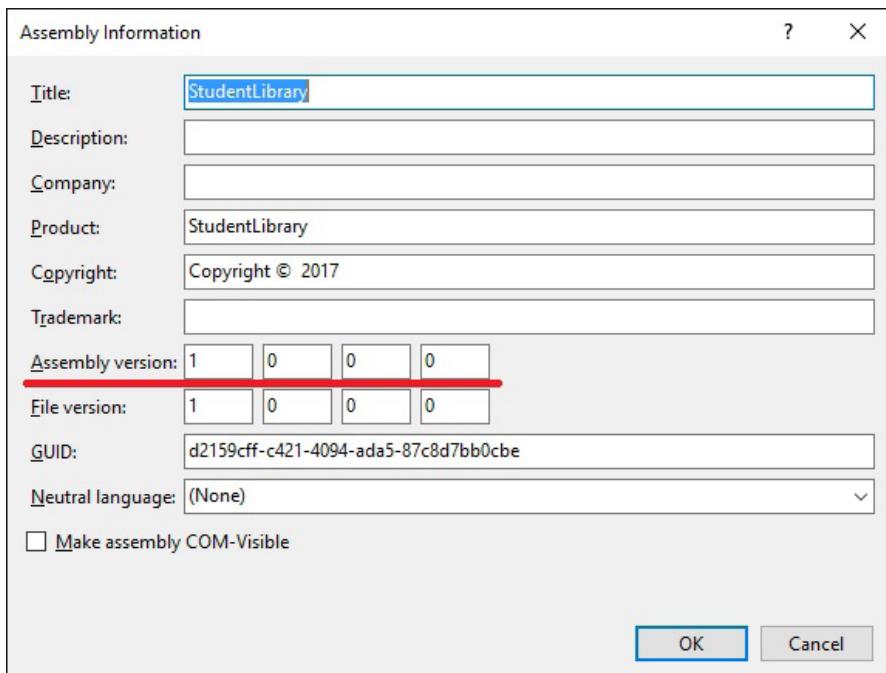
Для примеров этого раздела мы воспользуемся библиотекой StudentLibrary, которую создали в предыдущем разделе. Для того чтобы указать версию библиотеки необходимо открыть окно свойств проекта и выбрать в нем вкладку Application, а на ней нажать кнопку Assembly Information... (Рисунок 7.1).



**Рисунок 7.1. Вкладка Application**

В открывшемся окне Assembly Information вы можете указать различную информацию о текущей сборке,

в том числе и версию сборки, которая состоит из четырех чисел (слева на право): основная версия (*major*), подверсия (*minor*), номер полной компиляции (*build*) и номер ревизии для текущей компиляции (*revision*) (Рисунок 7.2).

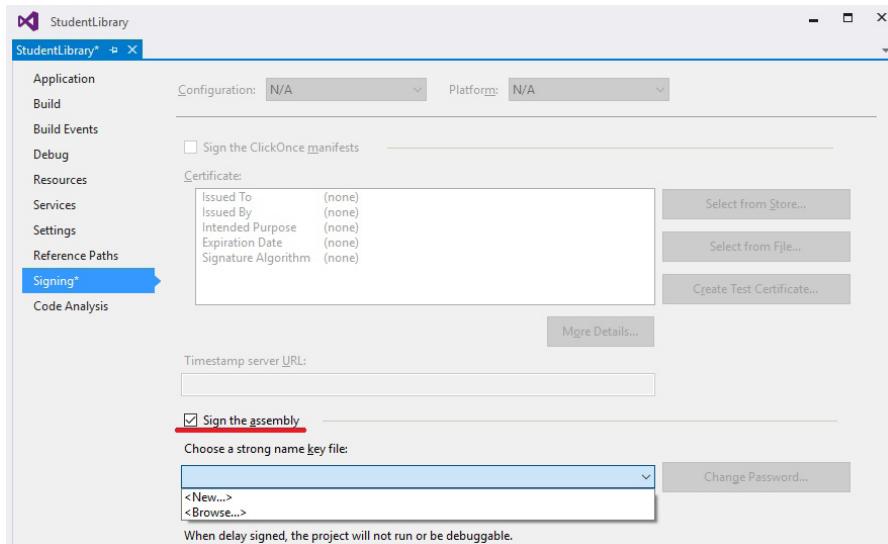


**Рисунок 7.2. Окно Assembly Information**

Первые два значения являются основными, и именно они используются при поиске необходимой DLL. Если найдено несколько библиотек с одинаковыми основными версиями, то происходит поиск по подверсии, и только если эти значения будут совпадать, то осуществляется поиск по номерам полной и текущей компиляции. Когда вы вносите значительные изменения в созданной

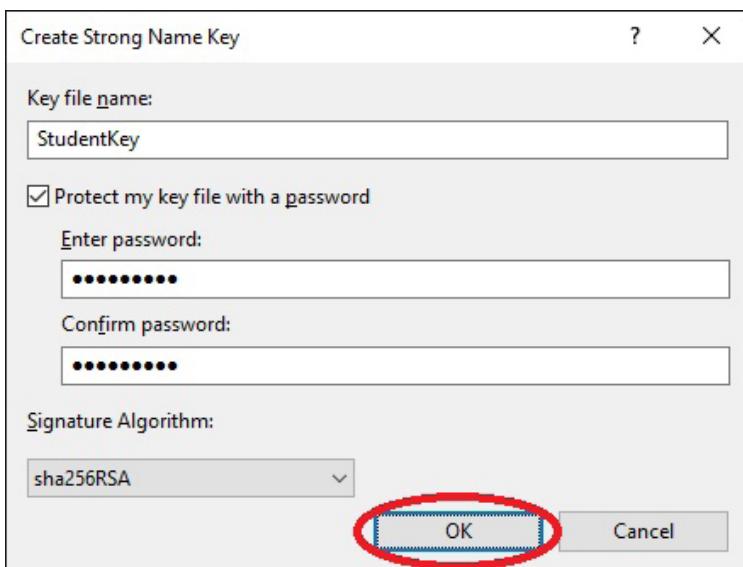
библиотеке, то следует изменять значения основной версии и подверсии, если осуществляете небольшие корректировки, то рекомендуется изменять только номера полной или текущей компиляции в зависимости от изменений.

Строгое имя библиотеки DLL гарантирует ее уникальность и может быть сформировано средствами Visual Studio 2015. Для этого также предназначено окно свойств проекта, но в этом случае необходимо выбрать вкладку Signing, а на ней установить флажок напротив пункта Sign the assembly. После чего станет доступным выпадающий список Choose a strong name key file, в котором вы можете выбрать существующий файл ключа со строгим именем или создать новый, пункты <Browse...> и <New...> соответственно (Рисунок 7.3).



**Рисунок 7.3. Вкладка Signing**

Для создания нового секретного ключа необходимо выбрать пункт <New...>, в результате чего появится окно Create Strong Name Key, где вы должны ввести имя файла и дважды пароль, которым будет защищена сборка, также вы можете выбрать один из двух алгоритмов шифрования, после чего нажать кнопку OK (Рисунок 7.4).

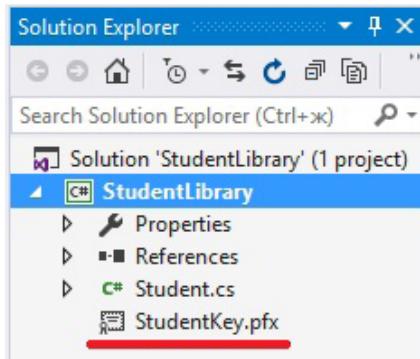


**Рисунок 7.4.** Окно Create Strong Name Key

Следует понимать, что шифрование защищает только строгое имя и не защищает вашу библиотеку от взлома и анализа ее кода.

После выполнения описанных выше действий, в проекте вашей библиотеки появится файл строгого ключа с расширением .pfx (Рисунок 7.5).

Вашу библиотеку необходимо перекомпилировать, после чего вы сможете поместить ее в GAC.



**Рисунок 7.5. Файл строгого ключа**

## Инсталляция сборок в GAC

Для того чтобы разместить полученную библиотеку в GAC, достаточно скопировать ее в специальный каталог (C:\WINDOWS\Microsoft.NET\Assembly\GAC\_MSIL). Однако предпочтительнее инсталлировать сборку в GAC при помощи утилиты командной строки gacutil.exe. Для этого необходимо открыть окно командной строки Visual Studio 2015 от имени администратора и перейти в каталог с требуемой библиотекой, затем установить сборку в GAC, воспользовавшись командой /i утилиты gacutil.exe. В случае успешного добавления сборки в GAC вы увидите соответствующее сообщение (Рисунок 7.6).

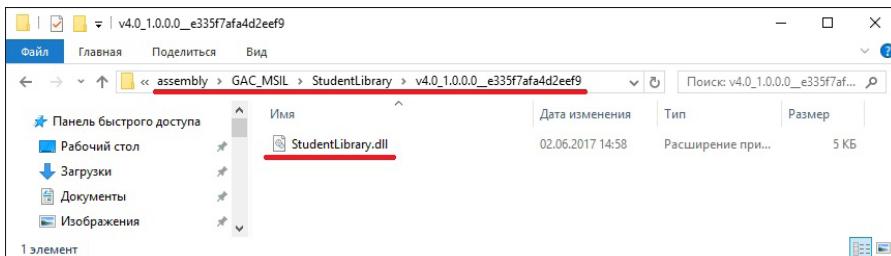
```
Administrator: Developer Command Prompt for VS2015
C:\WINDOWS\system32>cd C:\Users\Юрий\Documents\Visual Studio 2015\Projects\StudentLibrary\StudentLibrary\bin\Release
C:\Users\Юрий\Documents\Visual Studio 2015\Projects\StudentLibrary\StudentLibrary\bin\Release>gacutil /i StudentLibrary.dll
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
с Корпорацией Майкрософт (Microsoft Corporation). Все права защищены.

Сборка успешно добавлена в кэш

C:\Users\Юрий\Documents\Visual Studio 2015\Projects\Studentlibrary\StudentLibrary\bin\Release>
```

**Рисунок 7.6. Использование утилиты gacutil.exe**

Вы можете убедиться в том, что библиотека инсталлирована в GAC, открыв в проводнике Windows каталог C:\WINDOWS\Microsoft.NET\assembly\GAC\_MSIL, где вы увидите, что для вашей сборки создан отдельный подкаталог, совпадающий с названием вашей библиотеки, и в нем находится ваша библиотека DLL (Рисунок 7.7).



**Рисунок 7.7.** Содержимое GAC в проводнике Windows

После этого вы можете подключить DLL к различным проектам, так как это было описано в шестом разделе.

# 8. Класс Assembly

В процессе создания приложения вы определяете сборки, которые будут в нем использоваться, но существует еще возможность загружать сборки динамически. Для этих целей предназначен класс `Assembly` из пространства имен `System.Reflection`. Также в этом классе содержится ряд методов и свойств, которые позволяют получить информацию о сборках, например метод `GetName()` и свойство `FullName` возвращают имя сборки, а при помощи метода `GetModules()` можно получить все модули этой сборки. Для того чтобы выявить типы, определенные в сборке следует воспользоваться методом `GetTypes()`, создать экземпляры этих типов можно при помощи метода `GetType()`. Более подробную информацию об элементах этого класса вы можете получить, обратившись в MSDN.

Следующий пример демонстрирует использование класса `Assembly` для получения информации о текущей сборке (Рисунок 8.1).

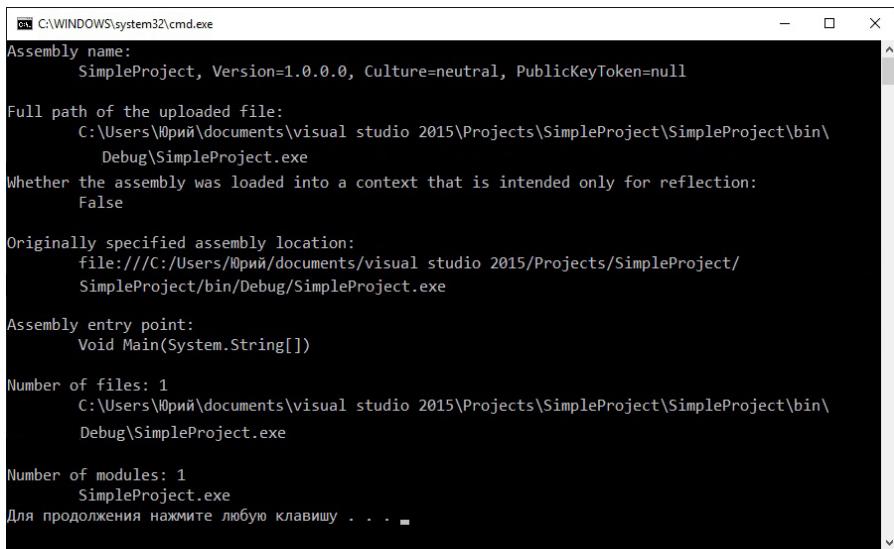
```
using System.IO;
using System.Reflection;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
// определение текущей сборки
Assembly theAssembly =
    Assembly.GetExecutingAssembly();

// использование свойств и методов
// для текущей сборки
WriteLine($"Assembly
    name:\n\t{theAssembly.FullName}\n");
WriteLine($"Full path of the uploaded
    file:\n\t{theAssembly.Location}\n");
WriteLine($"Whether the assembly was loaded
    into a context that is intended only
    for reflection:\n\t{theAssembly.
    ReflectionOnly}\n");
WriteLine($"Originally specified assembly
    location:\n\t{theAssembly.
    CodeBase}\n");
WriteLine($"Assembly entry
    point:\n\t{theAssembly.EntryPoint}");

FileStream[] files = theAssembly.GetFiles(true);
WriteLine($"nNumber of files: {files.Length}");
foreach (FileStream f in files)
{
    WriteLine($"t{f.Name}");
}
Module[] modules = theAssembly.
    GetLoadedModules(true);
WriteLine($"nNumber of modules:
    {modules.Length}");
foreach (Module m in modules)
{
    WriteLine($"t{m.Name}");
}
}
```



```
cmd C:\WINDOWS\system32\cmd.exe
Assembly name:
  SimpleProject, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Full path of the uploaded file:
  C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\bin\
    Debug\SimpleProject.exe
Whether the assembly was loaded into a context that is intended only for reflection:
  False
Originally specified assembly location:
  file:///C:/Users/Юрий/documents/visual studio 2015/Projects/SimpleProject/
    SimpleProject/bin/Debug/SimpleProject.exe
Assembly entry point:
  Void Main(System.String[])
Number of files: 1
  C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\bin\
    Debug\SimpleProject.exe
Number of modules: 1
  SimpleProject.exe
Для продолжения нажмите любую клавишу . . . =
```

**Рисунок 8.1.** Пример использования  
класса Assembly

## Загрузка сборок

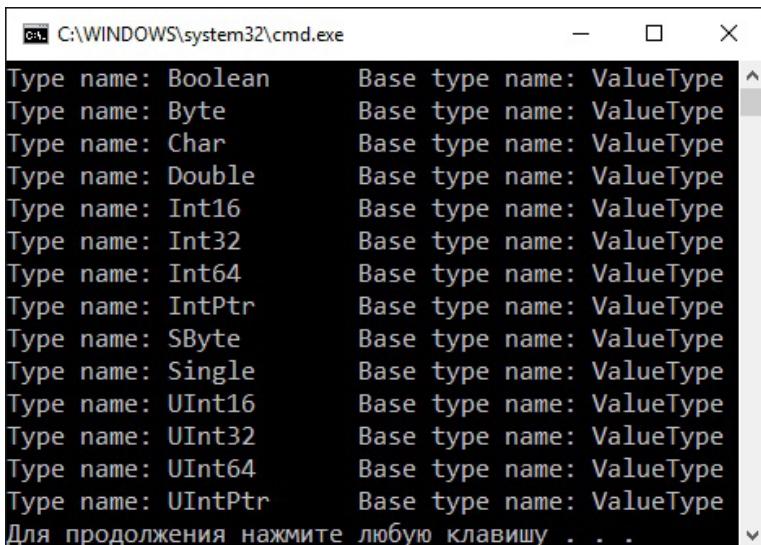
Загрузка сборок осуществляется при помощи вызова методов `Load()` или `LoadFrom()` класса `Assembly`. Метод `Load()` позволяет загрузить сборку по имени, причем может быть задана длинная форма имени сборки (простое имя, версия, язык и региональные параметры, маркер открытого ключа). При вызове метода `LoadFrom()` для загрузки сборки необходимо передать имя сборки или путь к файлу, содержащего манифест необходимой сборки. Существует ряд перегрузок этих методов, принимающих различное количество параметров, за более подробной информацией рекомендуем вам обратиться в MSDN.

В следующем примере демонстрируется использование метода Load() для вывода информации о примитивных типах, которые определены в сборке mscorelib.dll — сборке, содержащей пространство имен System (Рисунок 8.2).

```
using System;
using System.Reflection;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly a = Assembly.Load("mscorlib");
            foreach (Type t in a.GetTypes())
            {
                if (!t.IsPrimitive)
                {
                    continue;
                }

                Write($"Type name: {t.Name} ");
                if (t.BaseType != null)
                {
                    Write($"\\Base type name:
{t.BaseType.Name}");
                }
                WriteLine();
            }
        }
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
Type name: Boolean      Base type name: ValueType
Type name: Byte          Base type name: ValueType
Type name: Char          Base type name: ValueType
Type name: Double         Base type name: ValueType
Type name: Int16          Base type name: ValueType
Type name: Int32          Base type name: ValueType
Type name: Int64          Base type name: ValueType
Type name: IntPtr         Base type name: ValueType
Type name: SByte          Base type name: ValueType
Type name: Single          Base type name: ValueType
Type name: UInt16          Base type name: ValueType
Type name: UInt32          Base type name: ValueType
Type name: UInt64          Base type name: ValueType
Type name: UIntPtr         Base type name: ValueType
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 8.2.** Пример использования класса метода Load()

В предыдущем коде для получения информации о типах используется класс `Type` из пространства имен `System`, который обеспечивает средства доступа к метаданным. Свойство `IsPrimitive` позволяет определить является ли текущий объект примитивным типом, при помощи свойства `BaseType` определяется базовый тип текущего объекта.

В следующем примере мы будем использовать метод `LoadFrom()` и продемонстрируем создание приложения с поддержкой подключаемых компонентов. Для создания подключаемых компонентов необходимо создать интерфейс, который эти компоненты должны будут реализовать. Описание этого интерфейса следует выполнить в отдельной сборке и выдать ее разработчикам компонентов, которые будут наследовать свои компоненты

от этого интерфейса. Для использования компонентов выполняется их динамическая загрузка, приведение к типу интерфейса и вызов методов этого интерфейса.

Первым этапом необходимо создать библиотеку, например AssemblyExample, с описанием необходимого интерфейса, допустим `IAssemblyExample`, как это сделать рассматривалось в шестом разделе.

```
namespace AssemblyExample
{
    public interface IAssemblyExample
    {
        string SomeMethod(int n);
    }
}
```

Затем создадим библиотеку, которая будет играть роль модуля расширения, назовем ее ModuleOne. В этой библиотеке создадим класс `ClassOne`, который будет реализовывать интерфейс `IAssemblyExample` из библиотеки AssemblyExample, для чего последнюю необходимо подключить (см. раздел №6).

```
using AssemblyExample;

namespace ModuleOne
{
    public class ClassOne : IAssemblyExample
    {
        public string SomeMethod(int n)
        {
            return $"ModuleOne: {n}";
        }
    }
}
```

Как вы уже догадались, количество подключаемых модулей расширения может быть не ограничено, основное условие — классы в этих модулях должны реализовывать требуемый интерфейс `IAssemblyExample`. Мы предлагаем вам самостоятельно создать два или три модуля для закрепления полученного материала.

После этого вам необходимо создать новое консольное приложение, а все созданные вами сборки расширения скопировать в папку с исполняемым файлом этого приложения, например `bin\Debug`. Исходный код самого приложения находится ниже, обращаем ваше внимание на необходимость подключения библиотеки `AssemblyExample`.

```
using AssemblyExample;
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {

            try
            {
                string path = Path.GetDirectoryName(Assembly.
                    GetEntryAssembly().Location);
```

```
List<Type> list = new List<Type>();

foreach (string s in Directory.GetFiles(path,
    "*.dll"))
{
    Assembly a = Assembly.LoadFrom(s);
    foreach (Type t in a.GetExportedTypes())
    {
        if (!t.IsClass ||
            !typeof(IAssemblyExample).
            IsAssignableFrom(t))
        {
            continue;
        }
        list.Add(t);
    }
}

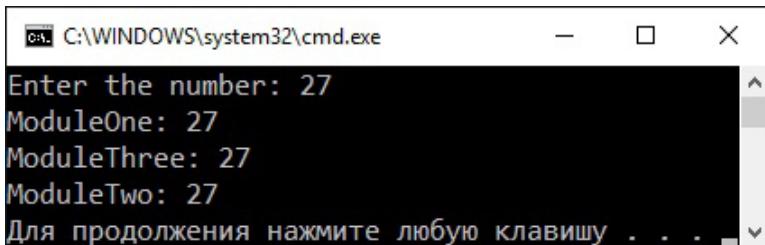
Write("Enter the number: ");
int number = int.Parse(ReadLine());

// вызов метода интерфейса для всех
// найденных типов
foreach (Type t in list)
{
    WriteLine((Activator.CreateInstance(t)
        as IAssemblyExample).SomeMethod(number));
}

catch (Exception ex)
{
    WriteLine(ex.Message);
}
}
```

В приведенном выше коде мы определяем имя каталога, из которого запущена сборка при помощи метода `GetDirectoryName()` класса `Path`. Затем получаем все файлы с расширением `.dll` в этом каталоге (метод `.GetFiles()` класса `Directory`). После этого получаем все открытые типы из указанной сборки при помощи метода `GetExportedTypes()` класса `Assembly`. Для каждого типа осуществляем проверку, является ли данный тип классом и поддерживает ли он интерфейс `IAssemblyExample` (свойство `IsClass` и метод `IsAssignableFrom()` класса `Type`, соответственно) и если это так, то записываем его в список. После чего на основе каждого типа из этого списка создаем объект, при помощи метода `CreateInstance()` класса `Activator`, приводим его к интерфейсу `IAssemblyExample` и вызываем метод `SomeMethod()`, передавая, введенное пользователем, число.

Если у вас создано несколько подключаемых модулей, то результат работы программы будет приблизительно такой, как на рисунке 8.3.



**Рисунок 8.3.** Пример работы приложения с поддержкой подключаемых компонентов

## 9. Механизмы безопасности в .Net Framework (Code Access Security)

Неограниченный доступ к системным ресурсам какого-либо компьютера, таким как локальные файлы, файлы в удаленной файловой системе, ключи реестра, базы данных и т.д. может привести к потенциальным угрозам безопасности. Потому что вредоносный код, запущенный на этом компьютере, может их повредить, например, удалить критические файлы, изменить ключи реестра или удалить данные, хранящихся в базах данных.

Для защиты компьютерных систем от вредоносного кода, платформа .NET Framework предоставляет технологию безопасности, разработанную для обеспечения защиты системных ресурсов при выполнении сборок .NET, которая называется безопасностью доступа к коду — Code Access Security (CAS).

CAS применяет политики безопасности в среде .NET, предотвращая несанкционированный доступ к защищенным ресурсам и операциям, и предназначена для решения проблем, возникающих при получении кода из внешних источников. Наличие в таком коде ошибок и уязвимостей приведет к возможности выполнения вредоносным кодом различных операций без ведома пользователя.

Каждая сборка содержит разрешения, определенные в политике безопасности, что формирует основу для

предоставления коду возможностей для выполнения необходимых действий. На основе этой информации CAS разрешает только те операции, которые пользовательский код может выполнить. Эта функция применима ко всему управляемому коду, ориентированному на CLR.

Безопасность доступа к коду платформы .NET Framework состоит из следующих частей:

- группа кодов — это логическая группировка кода с заданным условием принадлежности, таким как LocalIntranet или Internet;
- доказательство — связанная со сборкой информация: каталог приложения, издатель, URL и зона безопасности;
- политика безопасности — набор правил, настраиваемых администратором для определения разрешений для кода, выраженного в иерархическом порядке на четырех уровнях в качестве домена предприятия, машины, пользователя и приложения;
- разрешения — это основные права, необходимые для доступа к защищенному ресурсу или выполнения защищенной операции.

Разрешения можно задать при помощи кода двумя способами:

- декларативный режим реализуется путем определения атрибута `FileIOPermissionAttribute` на уровне сборки, класса или элемента класса, и используется, когда вызовы необходимо оценивать во время компиляции приложения (набор разрешений задается при помощи значений перечисления `SecurityAction`);

- императивный режим основан на выполнении методов класса `FileIOPermission`, используется, когда вызовы необходимо оценивать при выполнении кода (разрешения задаются при помощи значений перечисления `FileIOPermissionAccess`).

Разрешения на выполнение определенных операций вычисляются на основании набора разрешений в кодовых группах и на уровне политики безопасности. Среда CLR обеспечивает требуемые разрешения в предоставленных разрешениях текущей сборки, если разрешение не предоставлено, тогда будет сгенерировано исключение безопасности `SecurityException`.

Следующий пример демонстрирует использование класса `Assembly` для получения параметров безопасности нашей текущей сборки (Рисунок 9.1).

```
using System;
using System.Reflection;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        /// <summary>
        /// Returns the security settings for the assembly
        /// </summary>
        public static string GetCasSecurityAttributes()
        {
            Assembly a = Assembly.GetExecutingAssembly();

            StringBuilder sb = new StringBuilder();

```

```
sb.AppendFormat($"Security rule:  
{a.SecurityRuleSet}\n");  
  
sb.AppendFormat($"Fully trusted assembly:  
{a.IsFullyTrusted}\n");  
  
// получение типа основного класса сборки  
Type t = a.GetType("SimpleProject.Program");  
  
sb.AppendFormat($"Whether the current type is  
security-critical:  
{t.IsSecurityCritical}\n");  
  
sb.AppendFormat($"Whether the current type  
is security-safe-critical at  
the current trust level:  
{t.IsSecuritySafeCritical}\n");  
  
sb.AppendFormat($"Whether the current  
type is transparent at  
the current trust level:  
{t.IsSecurityTransparent}\n");  
  
try  
{  
    sb.AppendFormat($"{Environment.NewLine}Number of permissions  
of the current assembly:  
{a.PermissionSet.Count}\n");  
}  
  
catch (Exception ex)  
{  
    sb.AppendFormat($"{Environment.NewLine}{ex.Message}\n");  
}  
  
return sb.ToString();  
}
```

```
static void Main(string[] args)
{
    try
    {
        WriteLine(GetCasSecurityAttributes());
    }
    catch (Exception ex)
    {
        WriteLine(ex.Message);
    }
}
```

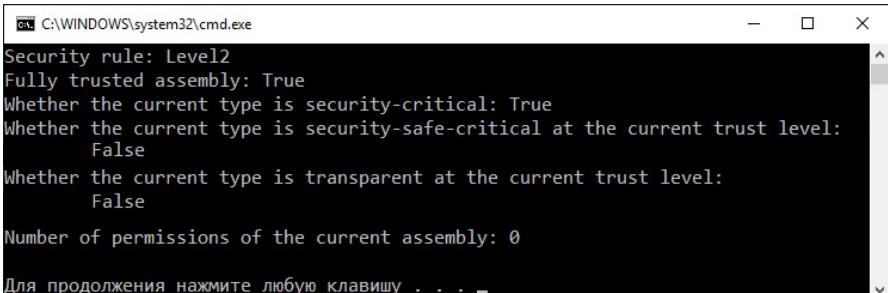
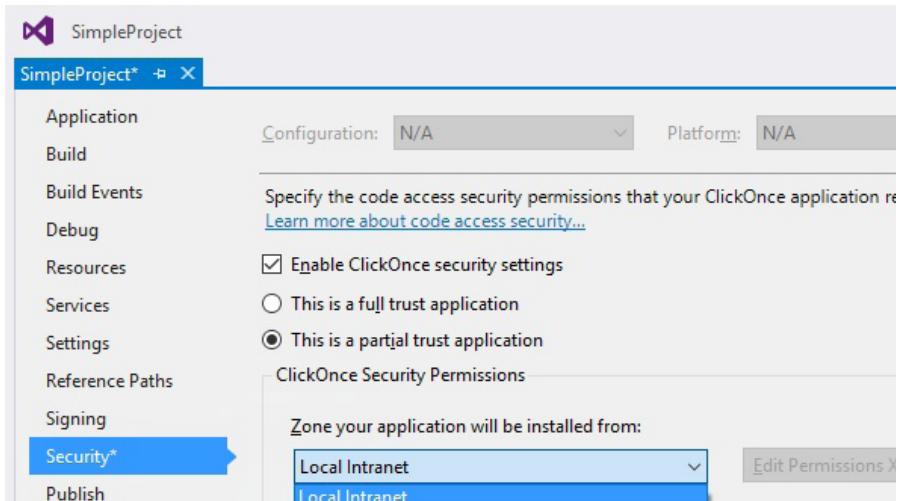


Рисунок 9.1 Параметры безопасности текущей сборки

Для того чтобы установить зоны безопасности для запуска приложения можно воспользоваться вкладкой Security окна свойств текущего приложения. Установив флажок «Enable ClickOnce security settings», вы сможете выбрать, будет ли приложение работать на клиентском компьютере с полным или частичным доверием. При выборе частичного доверия к приложению существует возможность указать дополнительные настройки (Рисунок 9.2).

## 9. Механизмы безопасности в .Net Framework...



**Рисунок 9.2.** Вкладка Security свойств приложения

## 10. Домашнее задание

---

Разработать приложение, позволяющее определить размер диагонали монитора текущего компьютера в дюймах.





## Урок №12

# Использование унаследованного программного кода, Code Access Security

© Юрий Задерей.

© Компьютерная Академия «Шаг»

[www.itstep.org](http://www.itstep.org).

Все права на охраняемые авторским правом фото-, аудио- и видеопропизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.