

Введение в C#

Основные сущности проекта .NET



Оглавление

[На этом уроке](#)

[Элементы проекта .NET](#)

[Целевая платформа проекта](#)

[Создание библиотеки классов](#)

[Свойства проекта .NET](#)

[Вкладка Application](#)

[Вкладка Build](#)

[Вкладка Build events](#)

[Вкладка Debug](#)

[Вкладка Resources](#)

[Вкладка Settings](#)

[Дополнительные инструменты отладки](#)

[Edit and Continue](#)

[Перемещение курсора \(в режиме отладки\)](#)

[Домашнее задание](#)

[Используемые источники](#)

На этом уроке

1. Рассмотрим основные составляющие проекта .NET.
2. Рассмотрим конфигурации сборки, директивы компиляции и их назначение.
3. Изучим некоторые дополнительные инструменты отладки.

Элементы проекта .NET

Целевая платформа проекта

Программы, написанные на платформе .NET могут быть скомпилированы с указанием разрядности целевой платформы. Этот показатель задаёт разрядность среды CLR, используемой при запуске программы. В настоящее время широко распространены процессоры, которые имеют 32- или 64-битную архитектуру.

Перед тем как мы рассмотрим целевые платформы .NET, сравним различия между 32- и 64-битными процессорами и операционными системами. Например, 64-битные процессоры и операционные системы пришли на смену 32-битным и привнесли с собой ряд преимуществ:

1. Некоторые 32-битные операционные системы имели [ограничение](#) по количеству управляемой оперативной памяти в 3 ГБ. Даже если устройство содержало больший объём памяти, пользователю было доступно только 3ГБ (3.5 в Windows XP).

Другие операционные системы имели возможность управлять до 64ГБ оперативной памяти. Но даже такое значение кажется ничтожно малым на фоне возможностей 64-битных операционных систем. Для сравнения можно рассмотреть современный серверный компьютер Mac Pro с поддержкой до 1.5 ТБ оперативной памяти (1536 ГБ).

2. Некоторые 32-битные операционные системы и приложения подвержены серьёзной ошибке, имеющей название «[Проблема 2038 года](#)». Это может привести к сбоям 32-битной техники в 2038 году по причине переполнения представления дат и времени. В результате этого, после 19.01.2038 произойдёт обнуление времени. Оно установит на устройствах под управлением 32-битного ПО 1901 год.

64-битные операционные системы и приложения не подвержены такой проблеме. Справедливости ради стоит отметить, что проблема уже решена в ядре наиболее популярной операционной системы Linux (при учёте всей компьютерной техники). Но многие 32-битные приложения продолжают содержать эту ошибку.

Остаётся надеяться, что к 2038 году все компьютеры перейдут на 64-битное программное обеспечение.

Платформа .NET позволяет выбрать в качестве целевой платформы один из трёх вариантов: Any CPU, x64, x86.

Названия первых 32-битных процессоров от Intel оканчивались двумя цифрами — 8 и 6. Поэтому название x86 закрепилось за 32-битной архитектурой процессоров.

По большому счёту разработка на платформе .NET освобождает от необходимости поддерживать в исходном коде ту или иную архитектуру. В большинстве случаев мы можем поставлять сборки в конфигурации Any CPU и не предпринимать никаких лишних действий. Но в некоторых ситуациях нам всё же может понадобиться архитектура x86 / x64. Это зависит от специфики разрабатываемого приложения.

Если нужно хранить в памяти и обрабатывать большие объёмы данных, стоит посмотреть в сторону компиляции в x64. Напротив, 32-битные приложения, как правило, используют память компьютера более экономно.

Отдельно стоит отметить, что при разработке приложений, которые используют сторонние библиотеки (например, драйверы), придётся брать в расчёт разрядность таких библиотек. Рассмотрим, как поведёт себя среда исполнения в зависимости от разных сочетаний разрядности ОС.

При запуске под 32-битной операционной системой:

- Any CPU: приложение запускается как 32-битное. Может использовать библиотеки, скомпилированные в конфигурации Any CPU / x86. При попытке использовать 64-битную библиотеку произойдёт исключительная ситуация `BadImageFormatException`;
- x86: поведение такое же, как у Any CPU;
- x64: приложение не будет совместимо с 32-битной ОС, при запуске произойдёт исключение `BadImageFormatException`.

При запуске под 64-битной операционной системой:

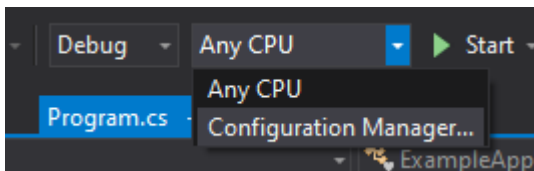
- Any CPU: приложение запускается как 64-битное. Может использовать библиотеки, скомпилированные в конфигурации Any CPU / x64. При попытке использовать 32-битную библиотеку произойдёт исключительная ситуация `BadImageFormatException`;
- x64: поведение такое же, как у Any CPU;
- x86: приложение запускается как 32-битное. Может использовать библиотеки, скомпилированные в конфигурации Any CPU / x86. При попытке использовать 64-битную библиотеку произойдёт исключительная ситуация `BadImageFormatException`.

Из приведённых данных можно сделать два вывода:

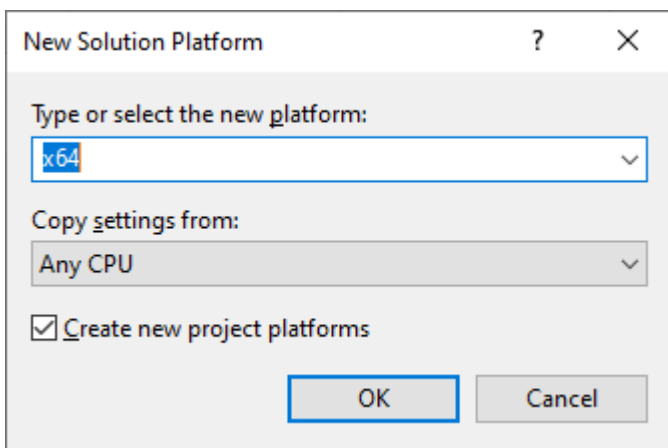
1. В рамках одного приложения можно использовать только библиотеки той же разрядности.
2. 64-битные операционные системы способны запускать 32-битные приложения.

Рассмотрим процесс добавления целевой платформы в проект. Стоит отметить, что по умолчанию новая целевая платформа добавляется в решение и во все находящиеся в нём проекты. Это сделано, чтобы упростить процесс сборки всего решения под разные платформы. Подробнее мы остановимся на этом в следующем разделе.

Выберем пункт Configuration Manager:



В открывшемся диалоге выберем в выпадающем списке Active solution platform, кликнем на пункт New....:



Нажмём ОК. Подобным образом добавим платформу x86. Теперь мы можем выбирать целевую архитектуру будущего приложения.

В следующем разделе рассмотрим создание и использование библиотек классов. Произведём попытку запустить приложение в совокупности с библиотекой, разрядности которых не совпадают.

Создание библиотеки классов

При разработке программного обеспечения практически всегда используются сторонние библиотеки (сборки). Даже мы в наших простых примерах формально используем сторонние библиотеки, хотя они и являются стандартными.

Например, для консольного ввода-вывода, используем библиотеку `mscorlib.dll`. Она поставляется вместе с .NET Framework. В противном случае нам бы пришлось самим описывать взаимодействие с системой для считывания кодов клавиатурных клавиш и вывода на экран.

Кроме стандартных библиотек, поставляемых с .NET, есть множество сторонних, решающих широкий спектр задач.

Можно создавать свои библиотеки классов для логического разделения опций приложения. Это важно и для переиспользования одного и того же кода в разных проектах.

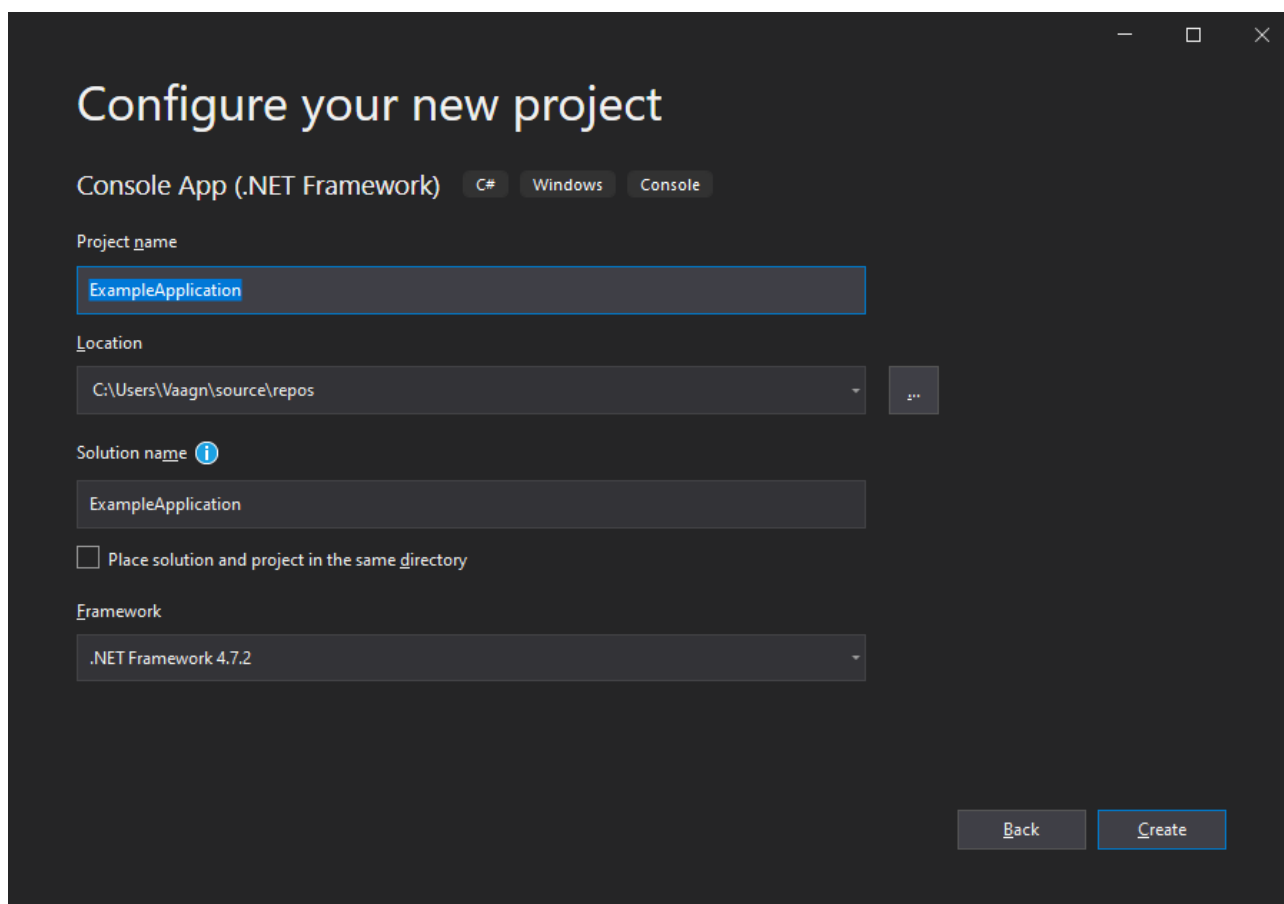
Библиотеки классов — термин мира .NET. Обычно такие библиотеки называются динамическими (Dynamic Link Library, .dll). Например, в работе с ОС Windows или совместно используемыми объектами (Shared Object, .so) в случае с Linux.

В обоих случаях это бинарные файлы, содержащие часть программного кода, который может быть использован в разных приложениях. Использование библиотек даёт разработчикам ряд преимуществ, например:

- можно обновлять отдельные библиотеки, а не всё приложение целиком;
- можно использовать одну и ту же библиотеку в нескольких приложениях.

Создадим консольное приложение, а затем добавим в решение проект библиотеки классов.

Сначала создадим новый проект консольного приложения, назовём его ExampleApplication:



Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

ExampleApplication

Location

C:\Users\Vaagn\source\repos

Solution name ⓘ

ExampleApplication

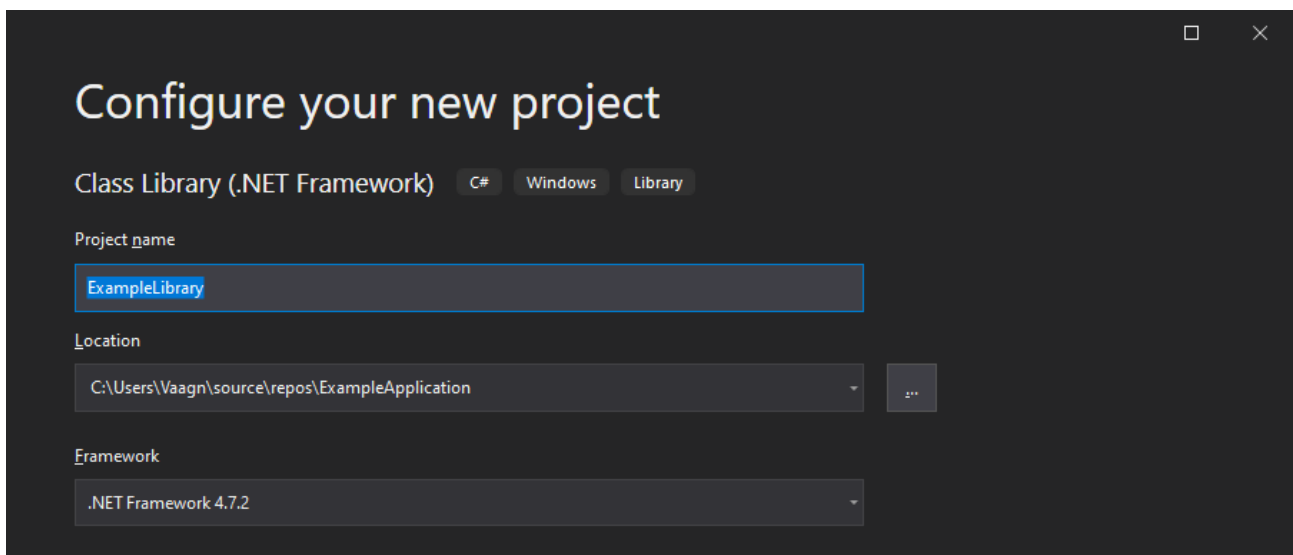
☐ Place solution and project in the same directory

Framework

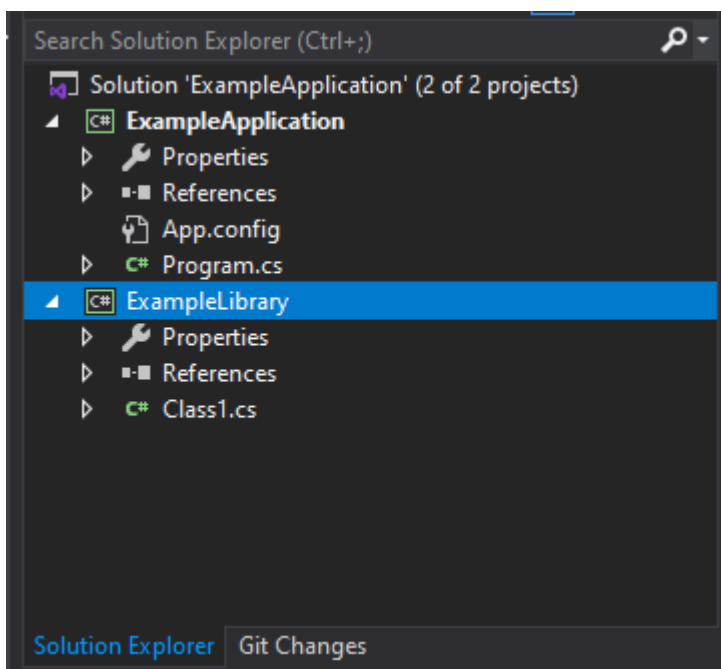
.NET Framework 4.7.2

Back Create

В обозревателе решений кликнем правой кнопкой мыши по [решению](#) и выберем пункт Add -> New Project. Далее в списке доступных шаблонов выберем шаблон библиотеки классов (Class Library) и создадим новый проект с названием ExampleLibrary:





Теперь наше решение содержит два проекта:



Когда запускаем сборку (`Ctrl + Shift + B`), Visual Studio собирает как проект приложения, так и библиотеки. Последняя на выходе образует бинарный файл с расширением `.dll`. При этом по умолчанию оба проекта собираются для одной и той же целевой платформы, которая задана в решении.

Для лучшего понимания результата сборки проверим выходную папку проекта `ExampleLibrary` после запуска сборки (файлы `.pdb` содержат отладочную информацию):

ExampleApplication > ExampleLibrary > bin > Debug			
Имя	Дата изменения	Тип	Размер
 ExampleLibrary.dll	12.12.2020 18:00	Расширение при...	4 КБ
 ExampleLibrary.pdb	12.12.2020 18:00	Program Debug D...	16 КБ

Добавим в проект библиотеки немного полезного кода и используем его в проекте приложения. Удалим созданный по умолчанию файл `Class1.cs`, добавим новый `Greeting.cs`. Его содержимое изначально будет таким: неиспользуемые пространства имён в начале файла удалены.

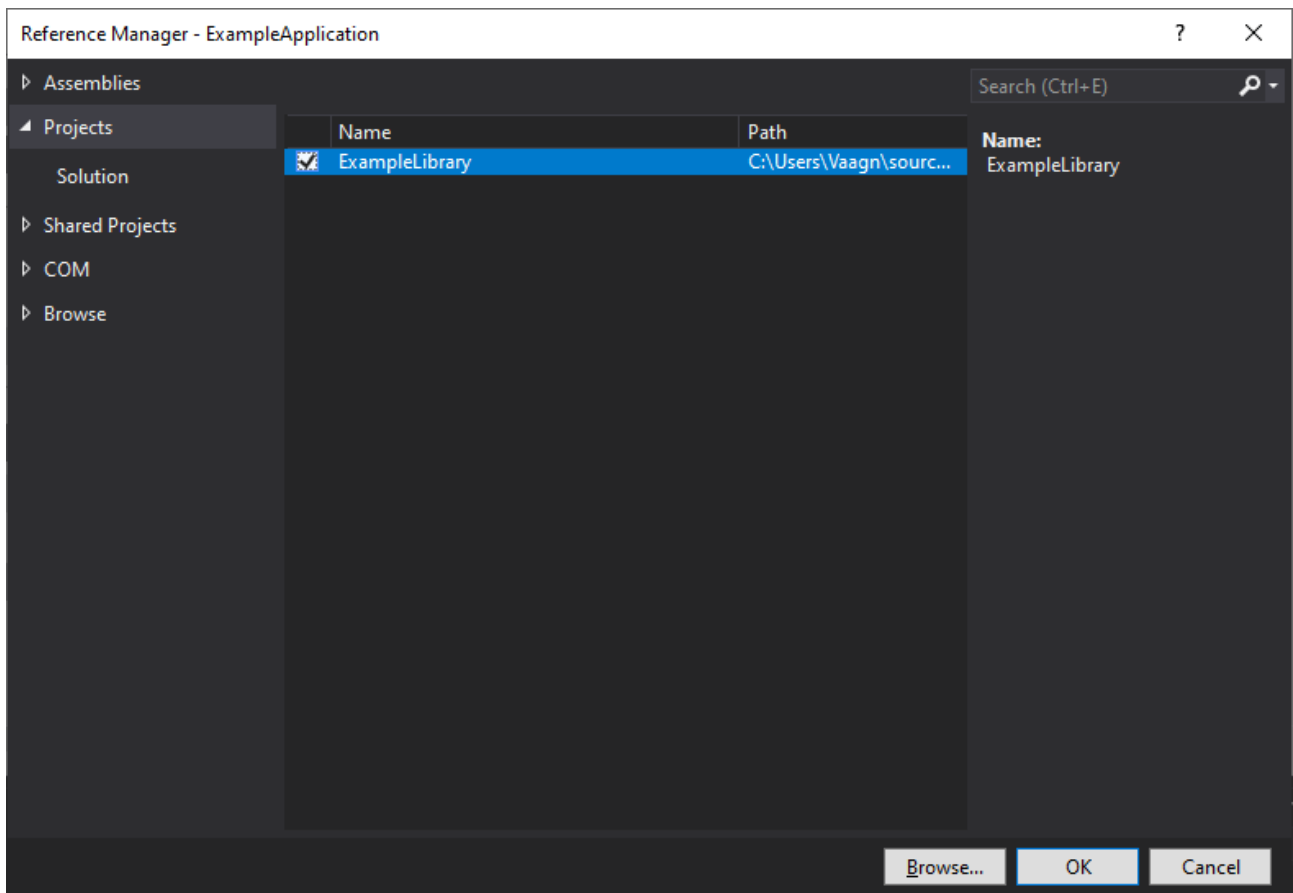
```
namespace ExampleLibrary
{
    class Greeting
    {
    }
}
```

Обратите внимание, что как и в случае с консольным приложением `ExampleApplication`, класс в библиотеке объявлен в одноимённом с проектом пространстве имён `ExampleLibrary`. Это происходит потому, что оно по умолчанию совпадает с именем проекта.

В следующем разделе мы вернёмся к этому пункту, а пока добавим в класс метод `SayHello`:

```
public static void SayHello()
{
    System.Console.WriteLine("Hello from ExampleLibrary!");
}
```

Далее используем класс `Greeting` из библиотеки в приложении. Для этого необходимо добавить в проект приложения ссылку на проект с библиотекой в том же окне, в котором добавляли ссылки на сборки .NET: `References -> Add Reference...` Затем слева нужно выбрать вкладку `Projects` и кликнуть на проект `ExampleLibrary`:



В настоящий момент класс `Greeting` не имеет модификатора `public`. Поэтому он будет доступен только в той сборке, в которой объявлен — в сборке библиотеки. Чтобы мы могли использовать этот класс в проекте приложения, добавим класс `Greeting` модификатору `public`:

```
public class Greeting
```

Теперь класс `Greeting` можно использовать в классе `Program`. Добавим соответствующее пространство имён и вызовем метод `Greeting.SayHello()`:

```
using ExampleLibrary;

namespace ExampleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Greeting.SayHello();
        }
    }
}
```

```
}  
  
}
```

После сборки проекта проверим выходную директорию консольного приложения. Наряду с другими файлами, директория содержит результат сборки проекта библиотеки классов:

ExampleApplication > ExampleApplication > bin > Debug

Имя	Дата изменения	Тип	Размер
ExampleApplication.exe	12.12.2020 19:02	Приложение	5 КБ
ExampleApplication.exe.config	12.12.2020 17:51	XML Configuratio...	1 КБ
ExampleApplication.pdb	12.12.2020 19:02	Program Debug D...	20 КБ
ExampleLibrary.dll	12.12.2020 19:02	Расширение при...	4 КБ
ExampleLibrary.pdb	12.12.2020 19:02	Program Debug D...	20 КБ

Результат выполнения программы:

```
Hello from ExampleLibrary!
```

В предыдущем разделе мы упоминали, что рассмотрим ситуацию, в которой приложение пытается загрузить библиотеку с несовместимой целевой платформой. Проверим настройки решения в Configuration Manager:

Configuration Manager?✕

Active solution configuration:Active solution platform:

Debugx86

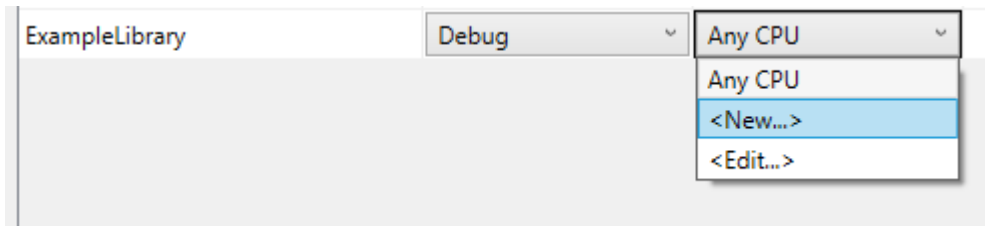
Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
ExampleApplication	Debug	x86	<input checked="" type="checkbox"/>	<input type="checkbox"/>
ExampleLibrary	Debug	Any CPU	<input checked="" type="checkbox"/>	<input type="checkbox"/>

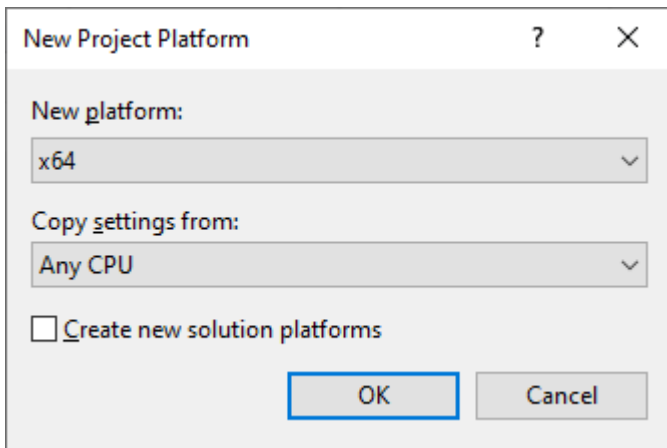
Когда мы добавляли в решение целевые платформы x86 / x64, проект ExampleLibrary ещё не был создан. Поэтому в нём настроена сборка только в Any CPU (как в любом новом проекте).

Так как Any CPU совместим и с 32-, и с 64-битными архитектурами, наше приложение сейчас полностью работоспособно. Можно собрать и запустить его в любой целевой платформе, при условии, что мы используем 64-битную ОС.

Для эксперимента добавим в проект целевые платформы x64 / x86. Для этого выберем элемент New для проекта ExampleLibrary:

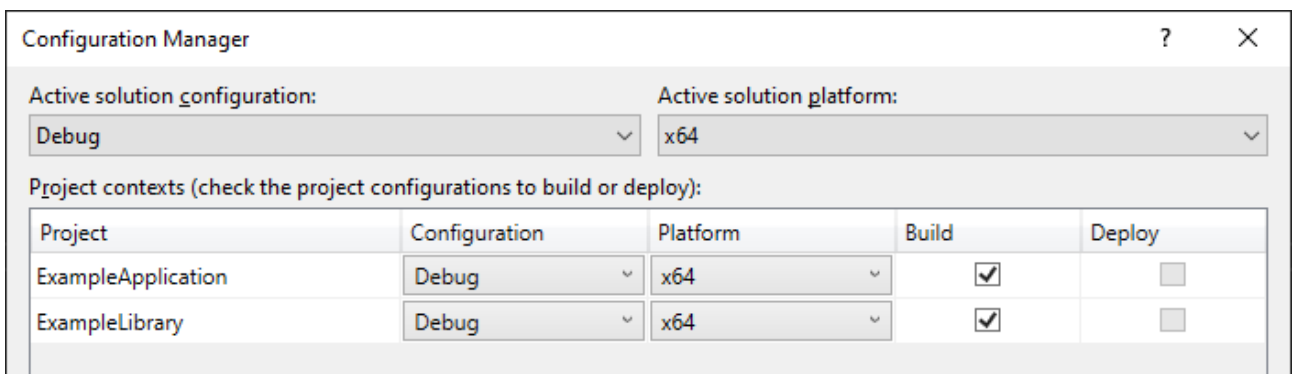


В следующем окне нужно снять флажок «Create new solution platforms», так как решение уже содержит все необходимые нам целевые платформы. Нужно лишь добавить их в проект:

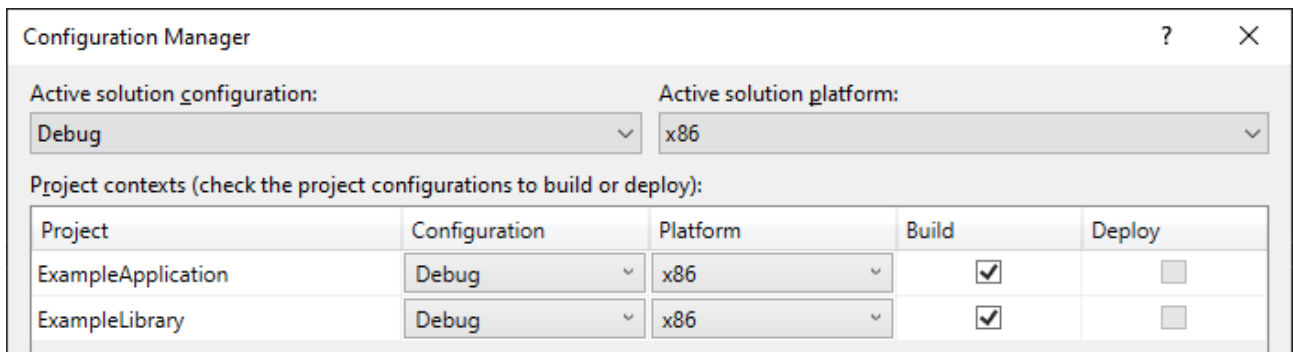


Повторим этот шаг для x86.

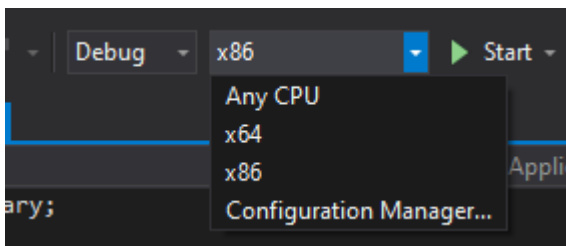
Далее проверим, что платформы решения и проектов совпадают. Для этого в выпадающем списке Active solution platform переберём все три платформы. Проверим, что у проектов указывается такая же платформа. Пример для x64:



И для x86:

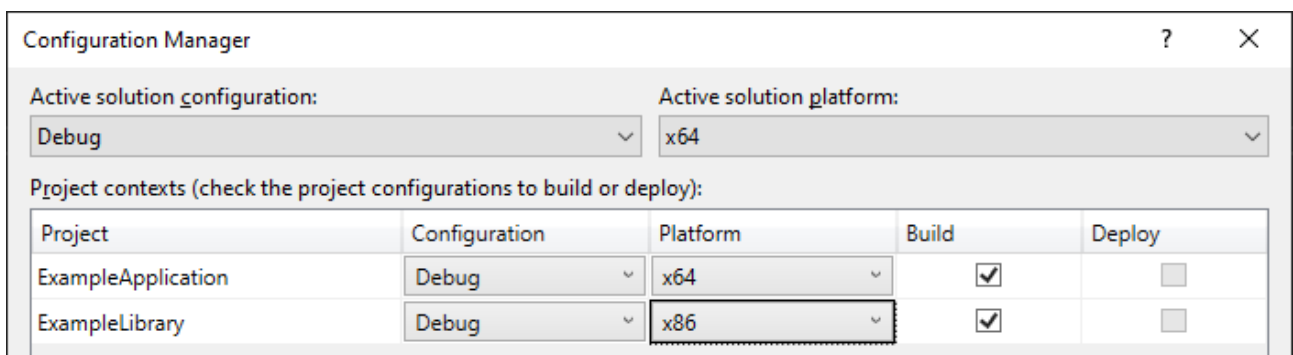


Теперь запустим приложение в каждой платформе и убедимся, что всё работает. Выберем поочередно Any CPU, x64, x86 из выпадающего списка и запустим приложение сочетанием клавиш Ctrl + F5:

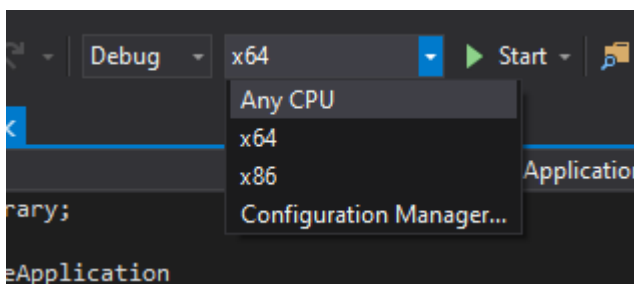


И наконец, воссоздадим ситуацию, когда возникает исключительная ситуация `BadImageFormatException`. Для этого приложение и библиотека должны иметь разные целевые платформы.

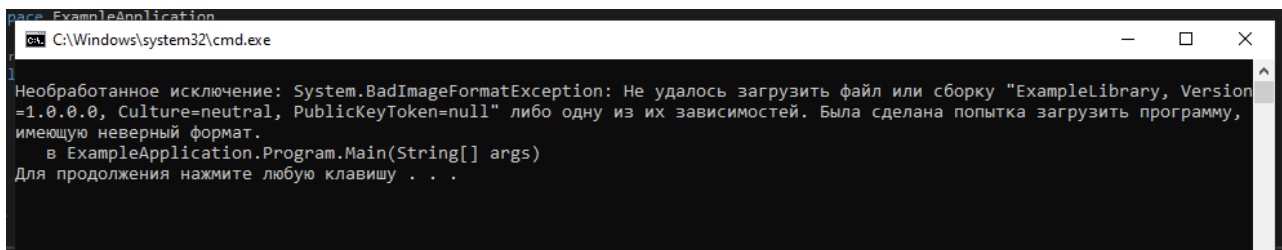
Рассмотрим ситуацию, в которой приложение будет 64-битным, а используемая в нём библиотека 32-битной. Обратная ситуация будет равнозначной: 32-битное приложение и 64-битная библиотека. Для этого вновь откроем менеджер конфигураций и выберем платформу x64. Выставим для библиотеки платформу x86:



Выставим целевую платформу решения в x64:



И запустим приложение сочетанием клавиш Ctrl + F5:



Свойства проекта .NET

В этом разделе рассмотрим основные свойства проекта .NET. Они обладают большим набором настроек, которые применяются в тех или иных случаях. Разбор всех свойств занял бы продолжительное время. Нам пришлось бы разбираться во множестве новых терминов.

Поэтому рассмотрим наиболее интересные свойства, которые будут основываться на уже имеющихся у нас знаниях. Разберёмся и с хранением настроек приложения. Эти знания могут быть полезны в будущих курсах.

Чтобы открыть свойства проекта, нужно кликнуть по проекту правой кнопкой мыши и выбрать пункт Properties. Свойства разделены на несколько вкладок. Далее рассмотрим их по порядку.

Вкладка Application

Assembly name — имя сборки — файла с расширением .exe или .dll

Default namespace — пространство имён по умолчанию, которое задаётся для новых файлов в проекте.

Target framework — используемая версия .NET. Чем новее версия, тем больше возможностей предоставляют стандартные библиотеки.

Assembly information — кнопка вызывает окно. В нём можно указать дополнительную информацию о сборке, которая затем отображается в свойствах исполняемого файла:

Assembly Information

Title: Amazing App

Description: C# console application

Company: Geekbrains

Product: Amazing App

Copyright: Copyright © Geekbrains, 2020

Trademark:

Assembly version: 1 2 3 4

File version: 1 2 3 4

GUID: fb4a6df2-30c8-4b05-b85e-90676f2bc4df

Neutral language: (None)

☐ Make assembly COM-Visible

OK Cancel

Свойства: ExampleApplication.exe

Общие Совместимость Безопасность Подробно Предыдущие версии

Свойство	Значение
Описание	
Описание файла	Amazing App
Тип	Приложение
Версия файла	1.2.3.4
Название продукта	Amazing App
Версия продукта	1.2.3.4
Авторские права	Copyright © Geekbrains, 2020
Размер	4,00 КБ
Дата изменения	12.12.2020 22:12
Язык	Независимо от языка
Исходное имя файла	ExampleApplication.exe

[Удаление свойств и личной информации](#)

OK Отмена Применить

Вкладка Build

Output path — задаёт путь к выходной папке, в которой будут создаваться результаты сборки.

Вкладка Build events

На этой вкладке можно указать команды, которые будут выполнены до и после сборки проекта. Это позволяет автоматизировать выполнение рутинных вещей: копирование файлов, создание zip-архива и прочее.

Вкладка Debug

Command line arguments — позволяет задать входные аргументы для нашего приложения.

Вкладка Resources

Эта вкладка позволяет задать ресурсы, хранимые в приложении. К ним относятся изображения, файлы, строковые константы. Чаще всего ресурсы применяются в приложениях с графическим интерфейсом.

Вкладка Settings

Остановимся на этой вкладке подробнее. Она позволяет задать настройки приложения. Зачастую при разработке приложения возникает необходимость сохранить какие-либо сведения в файл для последующего использования. Это может быть что угодно, например, настройки пользователя и прочее.

Рассмотрим работу с настройками приложения на примере. Кликнем по надписи «This project does not contain a default settings file. Click here to create one». В результате появится табличное представление. Зададим несколько настроек:

	Name	Type	Scope	Value
	UserName	string	User	
	Greeting	string	Application	Hello
*				

Параметр `Scope` влияет на уровень описания каждой из настроек. Пользовательские настройки могут быть изменены во время работы приложения, в то время как настройки уровня приложения доступны только для чтения.

Заданные настройки хранятся в файле `app.config`, который во время сборки копируется в выходную папку проекта и называется как `%Название сборки%.exe.config` (например, `ExampleApplication.exe.config`).

После изменения пользовательских настроек они сохраняются в отдельном файле в профиле пользователя. После запуска среда выполнения проверяет наличие такого файла и считывает настройки из него. Если такого файла нет, используются настройки по умолчанию из файла `app.config`.

Рассмотрим пример с чтением и записью настроек приложения. Напишем программу, которая запоминает имя пользователя:

```
using System;

namespace ExampleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            if (string.IsNullOrEmpty(Properties.Settings.Default.UserName))
            {

                Console.WriteLine("Введите имя пользователя:");
                Properties.Settings.Default.UserName = Console.ReadLine();
                Properties.Settings.Default.Save();
            }
            string userName = Properties.Settings.Default.UserName;
            string greeting = Properties.Settings.Default.Greeting;
            Console.WriteLine($"{greeting}, {userName}!");
        }
    }
}
```

Ранее мы описали во вкладке `Settings` два параметра — `Greeting` и `UserName`. Visual Studio автоматически производит кодогенерацию (файл `Settings.Designer.cs`), благодаря которой можно обращаться к этим параметрам как к свойствам класса — `Properties.Settings.Default.UserName`.

Обратите внимание на вызов метода `Properties.Settings.Default.Save()`. Он сохраняет все настройки, у которых `Scope=User` в этот конфигурационный файл:

```
<Profile Directory>\<Company Name>\<App Name>_<Evidence Type>_<Evidence Hash>\<Version>\user.config
```

Значения `Company Name`, `App Name`, `Version` подставляются из указанных значений в свойствах проекта на вкладке `Application` в окне `Assembly Information`. Значения `Evidence Type` и `Evidence Hash` — хэш-значения, которые вычисляются на основе `Assembly Information`.

Пример итогового пути к файлу с пользовательскими настройками:

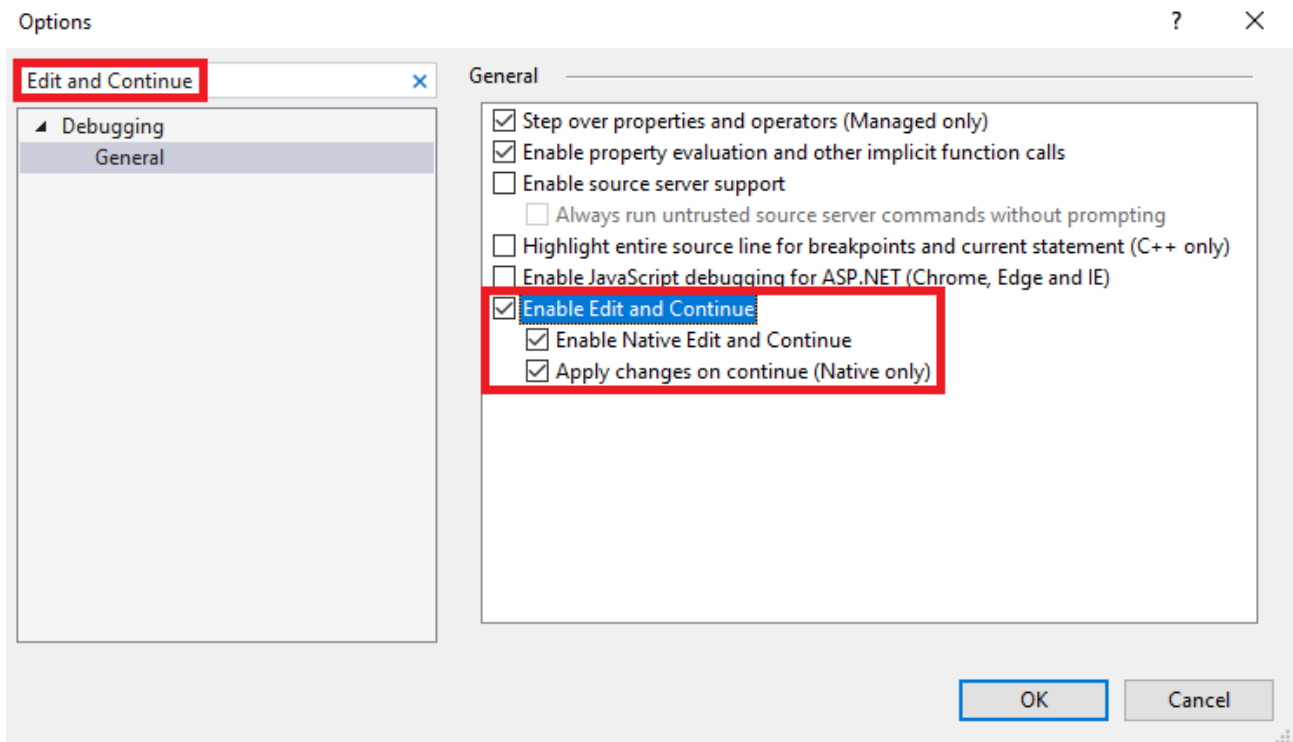
```
"C:\Users\TestUser\AppData\Local\Geekbrains\ExampleApplication.exe_Url_xcidg2y5py1qdl3hp0fmrdsgrls1gr1\1.2.3.4\user.config"
```

Дополнительные инструменты отладки

Edit and Continue

Edit and Continue — это режим отладки Visual Studio, который позволяет вносить изменения в исходный код во время отладки приложения. Это довольно удобно, особенно в тех случаях, когда после запуска приложения нам нужно осуществить пользовательский ввод или сделать сетевой запрос. Может понадобиться запрос к базе данных, и уже потом добраться до места, которое мы пытаемся отладить. Edit and Continue дают пройти все эти шаги один раз, а затем производить изменение кода «на месте», без перезапуска приложения.

Для начала убедимся, что эта опция включена. В главном меню Visual Studio выберем `Tools -> Options`, а в открывшемся окне введём `Edit and Continue` в строку поиска. Отобразятся следующие результаты:



Убедимся, что все флажки в этой группе включены.

Рассмотрим небольшой пример, демонстрирующий работу этой опции. У нас есть код, который по нашим ожиданиям должен вывести имя пользователя и возраст на экран. Но он ведёт себя не так, как мы предполагали. Возникает желание его отладить и разобраться в причинах. Вводить каждый раз имя пользователя, затем возраст — занятие не самое интересное.

Поэтому поставим точку останова на вызове метода `Console.WriteLine` и запустим приложение в режиме отладки:

```
static void Main()
{
    string name = Console.ReadLine();
    string age = Console.ReadLine();

    string result = $"{name} age";
    Console.WriteLine(result);
}
```

Введем нужные данные. Теперь мы попали на точку останова и можем посмотреть значение переменной `result`. Очевидно, забыли указать фигурные скобки вокруг слова `age`, поэтому на экран выводится просто `age`, а не значение одноименной переменной.

Сейчас мы уже знаем, в чём заключается ошибка, и как её исправить. Конечно, можно остановить отладку, изменить значение переменной `result`. Затем вновь запустить отладку, ввести данные в консоль и проверить вывод. Но мы можем упростить процесс отладки. Не останавливая отладку, добавим несколько операций в код, прямо во время отладки:

```
string name = Console.ReadLine();
string age = Console.ReadLine();

string result = $"{name} age";
Console.WriteLine(result); // мы стоим здесь на точке останова
result = $"{name} {age}";
Console.WriteLine(result);
```

Если просто исправить значение переменной `result`, то `Console.WriteLine` всё равно выведет текущее значение переменной — “`{name} age`”. Чтобы избежать повторного ввода данных, просто скопируем часть кода и внесём исправления. Нажмем трижды клавишу `F10`. Это действие выполнит текущую операцию `Console.WriteLine` и две следующих, а затем снова вернёт нас в режим отладки. В консоли приложения увидим следующий результат:

```
user
23
user age
user 23
```

Теперь можно остановить отладку и привести код в порядок.

Перемещение курсора (в режиме отладки)

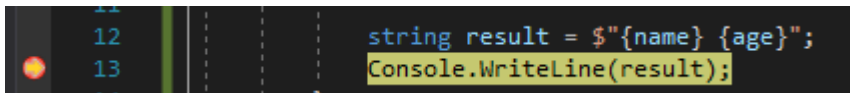
Режим `Edit and Continue`, который мы рассмотрели в предыдущем разделе — довольно мощный инструмент для отладки. Он может пригодиться в «глубокой отладке». Но в приведённом примере пришлось дублировать код и потом всё равно вносить в него правки после завершения. Гораздо удобнее, если бы после изменения кода, мы могли вернуться на несколько операций и выполнить их заново. К счастью, в `Visual Studio` есть такая возможность.

Возьмём пример из предыдущего пункта:

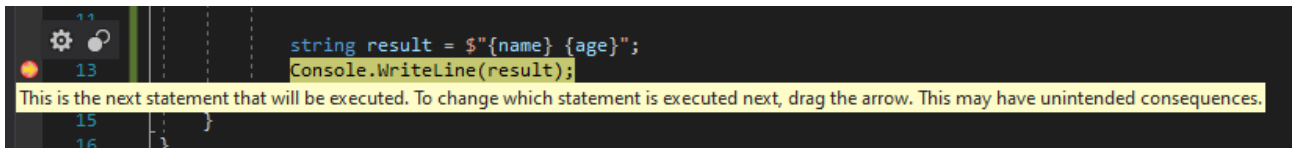
```
static void Main()
{
    string name = Console.ReadLine();
    string age = Console.ReadLine();

    string result = $"{name} age";
    Console.WriteLine(result);
}
```

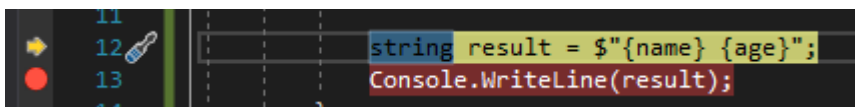
Поставим точку останова на вызове метода `Console.WriteLine` и запустим приложение в режиме отладки. Введём входные данные, попадём на точку останова и исправим значение, которое присваивается переменной `result`. Обратим внимание на то, как сейчас выглядит интерфейс Visual Studio на строке с точкой останова:



Слева отображается индикатор точки останова (красный круг), а также индикатор следующей операции (жёлтая стрелка). При наведении на эту область отобразится подсказка:



Давайте перенесём жёлтую стрелку на одну строку вверх:



Теперь следующая операция присвоит переменной `result` новое, корректное значение. Выполним две следующих операции, нажав дважды клавишу `F10`. Проверим вывод приложения:

```
user
23
user 23
```

Практическое задание

Создать консольное приложение, которое при старте выводит приветствие, записанное в настройках приложения (application-score). Запросить у пользователя имя, возраст и род деятельности, а затем сохранить данные в настройках. При следующем запуске отобразить эти сведения. Задать приложению версию и описание.

Используемые источники

1. [x86 — Википедия.](#)
2. [Страница приложения в свойствах проекта C# - Visual Studio.](#)
3. [Страница 'Построение' в конструкторе проектов \(C#\) - Visual Studio.](#)
4. [Страница 'Событий построения' в конструкторе проектов \(C#\) - Visual Studio.](#)
5. [Страница 'Отладка' в конструкторе проектов - Visual Studio.](#)
6. [Страница 'Параметры' в конструкторе проектов - Visual Studio.](#)