

Введение в C# и .NET

C# — объектно-ориентированный язык программирования общего назначения. Разработан в 1998-2001 годах группой инженеров компании Microsoft. C# работает на платформе **.NET**. Считается, что язык C# является ответом компании Microsoft на набравшую к тому времени большую популярность платформу Java компании Sun Microsystems (ныне принадлежит Oracle), около 75% его синтаксических возможностей аналогичны языку программирования Java.

Язык C# подходит для самых разных задач:

- **Консольные приложения**

Простые утилиты и скрипты с текстовым вводом-выводом.

- **Оконные (Desktop) приложения**

Классические Windows-программы и кроссплатформенные приложения (WinForms, WPF, MAUI, Avalonia).

- **Мобильные приложения**

Кроссплатформенная разработка под Android и iOS с помощью Xamarin или MAUI.

- **Веб-разработка**

Серверная логика и API (backend) на ASP.NET, а также интерактивные клиентские приложения на Blazor.

- **Игры**

Создание 2D/3D-игр в Unity, где C# используется для написания игровой логики и скриптов.

Для разработки на языке C# можно использовать:

- **Microsoft Visual Studio** — линейка систем разработки программного обеспечения от компании Microsoft. В своем составе имеют интегрированную среду разработки (IDE) и ряд других инструментов. Работает только на Windows. Скачать можно по [ссылке](#).
- **JetBrains Rider** — кроссплатформенная IDE для .NET от JetBrains, доступная на Windows, macOS и Linux. Основана на IntelliJ IDEA + ReSharper, предлагает мощный автодополнитель, рефакторинг и встроенную отладку. Скачать можно по [ссылке](#).

1 Платформа .NET

Платформа .NET — это унифицированная среда для разработки и запуска приложений на разных языках, созданная Microsoft. Она даёт единый набор инструментов и библиотек для работы с графикой, сетью, базами данных и другими задачами. На .NET можно писать на C#, F#, Visual Basic .NET и ряде других языков. Ключевые вехи её развития:

- **.NET Framework** (2002–2016) — Windows-ориентированная, закрытая платформа;
- **.NET Core** (2016–2020) — открытый исходный код, кроссплатформенность;
- **Единый .NET** (с .NET 5 в 2020 году и далее) с поддержкой Windows, Linux и macOS.

На сегодняшний день последней версией является **.NET 9**.

В основе .NET лежит **CLR (Common Language Runtime)** — среда исполнения, которая берёт **IL (Intermediate Language)** (промежуточный байт-код, получаемый при компиляции исходников) и превращает его в машинные инструкции. И **BCL (Base Class Library)** — единый набор готовых классов и функций для работы со строками, коллекциями, файловой системой, сетью и пр.

- **Компиляция** — преобразование исходного кода (C#, F# и т.п.) в IL или сразу в машинный код до выполнения; в отличие от **интерпретации**, где команды читаются и выполняются последовательно во время работы программы.

В .NET есть два основных подхода к превращению IL в нативный код:

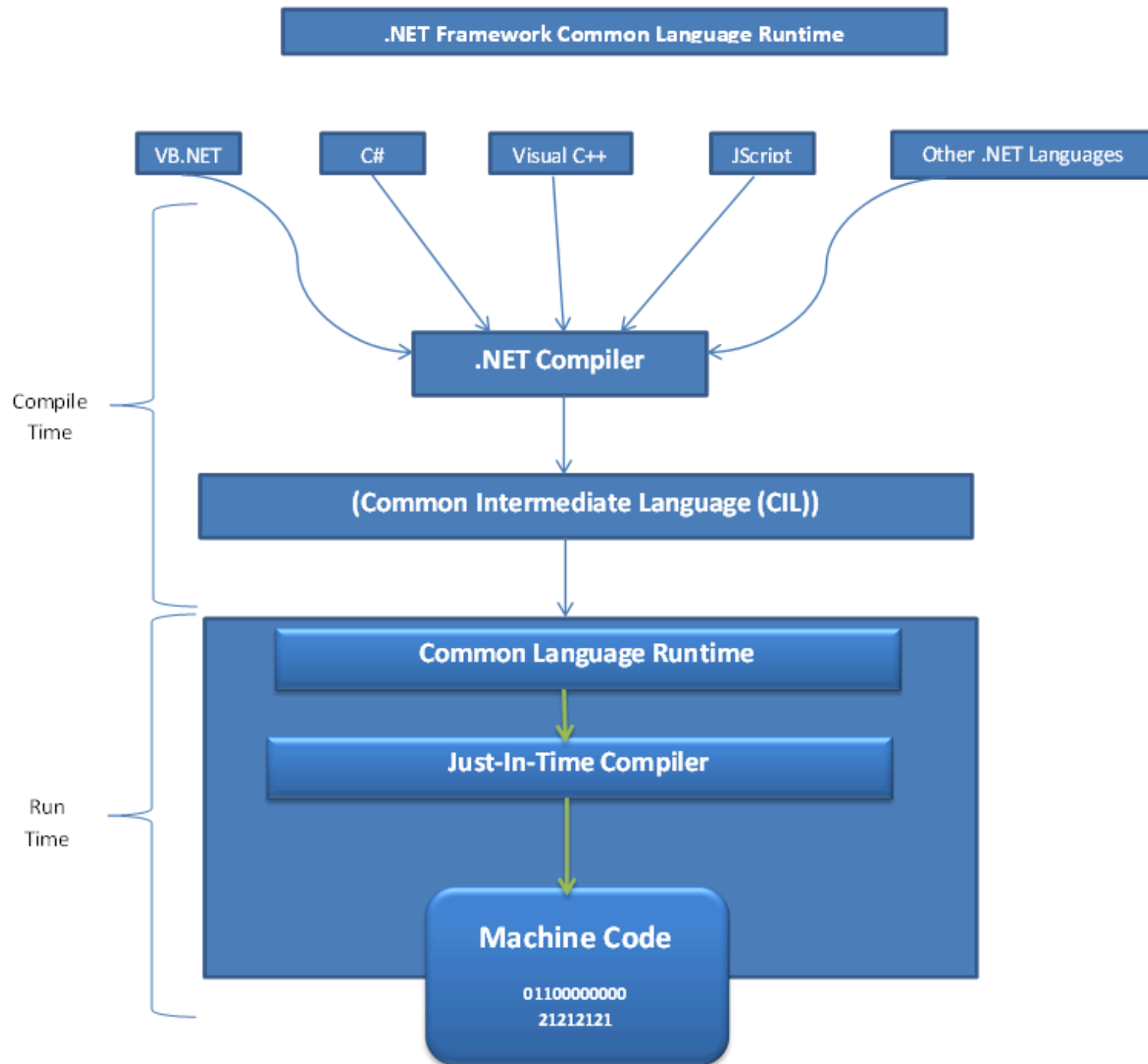
- **JIT (Just-In-Time)** — компиляция IL в машинный код «на лету» при загрузке частей программы, что позволяет оптимизировать её под конкретное железо. Используется по умолчанию и требует наличия установленного .NET.
- **AOT (Ahead-Of-Time)** — компиляция IL в машинный код заранее, на этапе сборки, встраивая рантайм-компоненты в итоговый файл. Приложение запускается быстрее и не зависит от установленного .NET.

Этапы жизненного цикла приложения:

- **Compile time** (время компиляции) — исходники (`.cs`) проверяются компилятором, превращаются в IL, собираются в `.dll` или `.exe` . Ошибки синтаксиса, типов или отсутствующих ссылок блокируют этот этап.
- **Run time** (время выполнения) — CLR загружает сборки, с помощью JIT/AOT получает машинный код, управляет памятью, проверяет безопасность типов и выполняет логику приложения. Ошибки, такие как деление на ноль, выход за границы массива или нехватки памяти, проявляются именно на этом этапе.

Платформа .NET состоит из двух частей:

- **.NET SDK (Software Development Kit)**
Набор инструментов для разработчика: CLI-команда (Command Line Interface) `dotnet` , компиляторы языков (C#, VB, F#), шаблоны проектов (`dotnet new`), управление пакетами (`dotnet add package`) и отладка. SDK устанавливают на машины, где пишут и тестируют код.
- **.NET Runtime**
Среда выполнения готовых приложений. Содержит CLR, JIT-компилятор, сборщик мусора и нужные библиотеки BCL, но не включает инструменты разработки. Runtime ставят на серверы и конечные компьютеры для запуска уже собранных приложений.

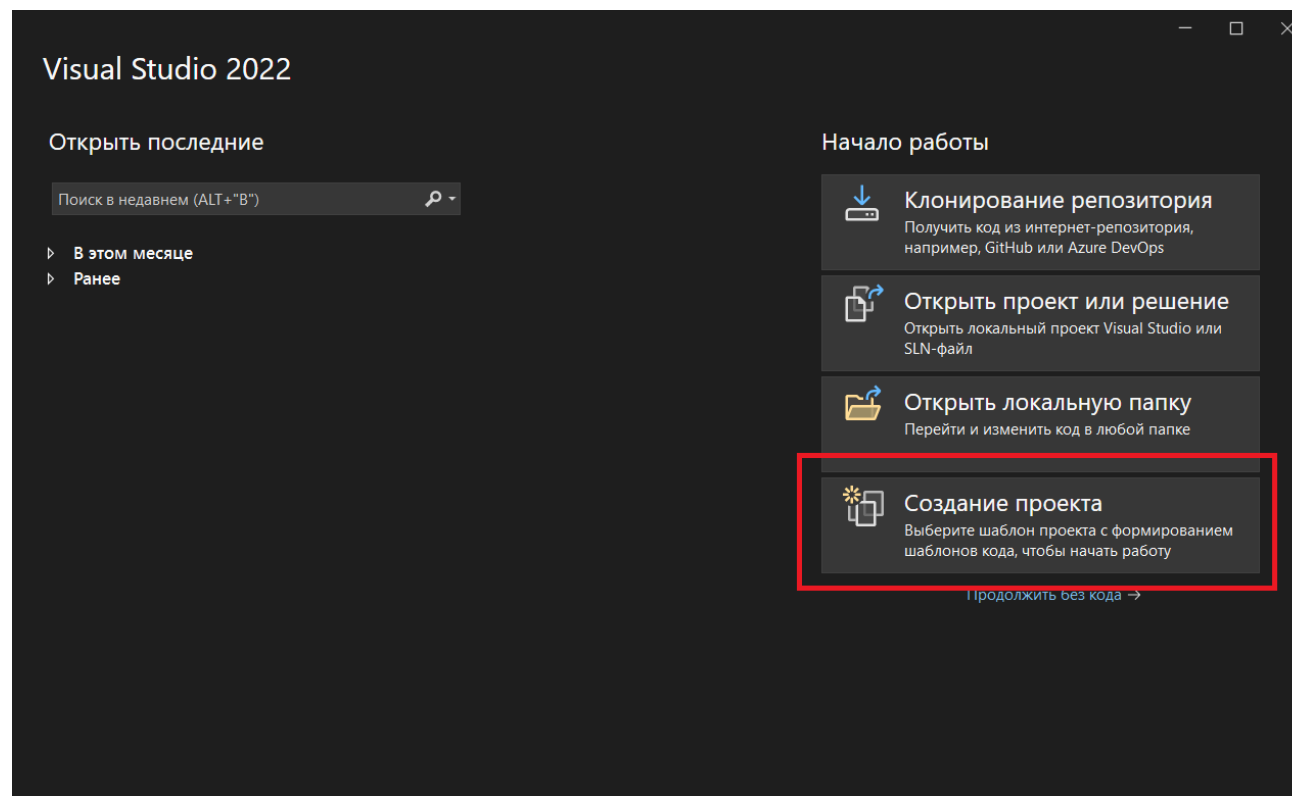


2 Пример приложения

Рассмотрим структуру проектов на C#, на примере простого консольного приложения, которые выводит *Hello, World!*.









Для создания проекта:

1. Запустите Visual Studio
2. Выберите **Создание проекта**
3. Выберите **Консольное приложение (Майкрософт)**
4. Укажите имя и расположение проекта
5. Выберите версию .NET



Создание проекта

Последние шаблоны проектов

 Веб-API ASP.NET Core (Майкрософт)	C#
 Веб-API ASP.NET Core (native AOT)	C#
 Пустой шаблон ASP.NET Core (Майкрософт)	C#
 Avalonia .NET MVVM App (AvaloniaUI)	C#
 Avalonia .NET App (AvaloniaUI)	C#
 Avalonia Cross Platform Application (AvaloniaUI)	C#
 Avalonia C# Project	C#
 Консольное приложение (Майкрософт)	C#

Поиск шаблонов (ALT+"B")



Очистить все

C#

Все платформы

Все типы проектов



Консольное приложение (Майкрософт)

Проект для создания приложения командной строки, которое может выполняться в среде .NET в Windows, Linux и macOS

C#

Linux

macOS

Windows

Консоль



Веб-приложение Blazor (Майкрософт)

Шаблон проекта для создания приложения Blazor, поддерживающего как отрисовку на стороне сервера, так и интерактивные возможности клиента. Этот шаблон можно использовать для веб-приложений с многофункциональными динамическими пользовательскими интерфейсами (UI).

C#

Linux

macOS

Windows

Blazor

Облако

Веб



Начальное приложение .NET Aspire (Майкрософт)

Шаблон проекта для создания приложения .NET Aspire с веб-интерфейсом Blazor и внутренней службой веб-API, при необходимости с использованием Redis для кэширования.

C#

.NET Aspire

API

Blazor

Облако

Common

Служба

Веб

Web API



Веб-приложение ASP.NET Core (Майкрософт)

Шаблон проекта для создания приложения ASP.NET Core с образцом

Назад

Далее

Настроить новый проект

Консольное приложение (Майкрософт)

C#

Linux

macOS

Windows

Консоль

Имя проекта

HelloWorld

Расположение

C:\Users\andro\Desktop\Новая папка

...

Имя решения ⓘ

HelloWorld



Поместить решение и проект в одном каталоге

Проект будет создан в "C:\Users\andro\Desktop\Новая папка\HelloWorld\"

Назад

Далее

Дополнительные сведения

Консольное приложение (Майкрософт)

C#

Linux

macOS

Windows

Консоль

Платформа ⓘ

.NET 8.0 (долгосрочная поддержка) ▾

☐ Включить поддержку контейнера ⓘ

ОС контейнера ⓘ

Linux ▾

Тип сборки контейнера ⓘ

Dockerfile ▾

☐ Не использовать операторы верхнего уровня ⓘ

☐ Включить публикацию AOT native ⓘ

Назад

Создать

ФайлПравкаВидGitПроектСборкаОтладкаТестАнализСредстваРасширенияОкноСправка

Поиск (Ctrl+Q)

HelloWorld

Вход

—

×

Live Share

Program.cs

HelloWorld

HelloWorld.Program

Main(string[] args)

```
1 namespace HelloWorld
2 {
3     Ссылка: 0
4     internal class Program
5     {
6         Ссылка: 0
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello, World!");
10        }
11    }
12 }
```

Обозреватель решений

Обозреватель решений — поиск (Ctrl+;)

Решение "HelloWorld" (1 проекта 1)

HelloWorld

Зависимости

Program.cs

100 %

Проблемы не найдены.

Стр: 7Симв: 48ПробелыCRLF

Вывод

Показать выходные данные из:

Список ошибок

Вывод

Обозреватель решений

Изменения Git

Готово

Добавить в систему управления версиями

Выбрать репозиторий

Ключевые понятия:

- **Файл с кодом** (`.cs`)

Отдельный файл на языке C#, содержащий классы, методы и другие элементы исходного кода.

- **Проект** (`.csproj`)

Совокупность кодовых файлов и настроек сборки, которая компилируется в **сборку** (программу или библиотеку).

- **Сборка** (`.exe` или `.dll`)

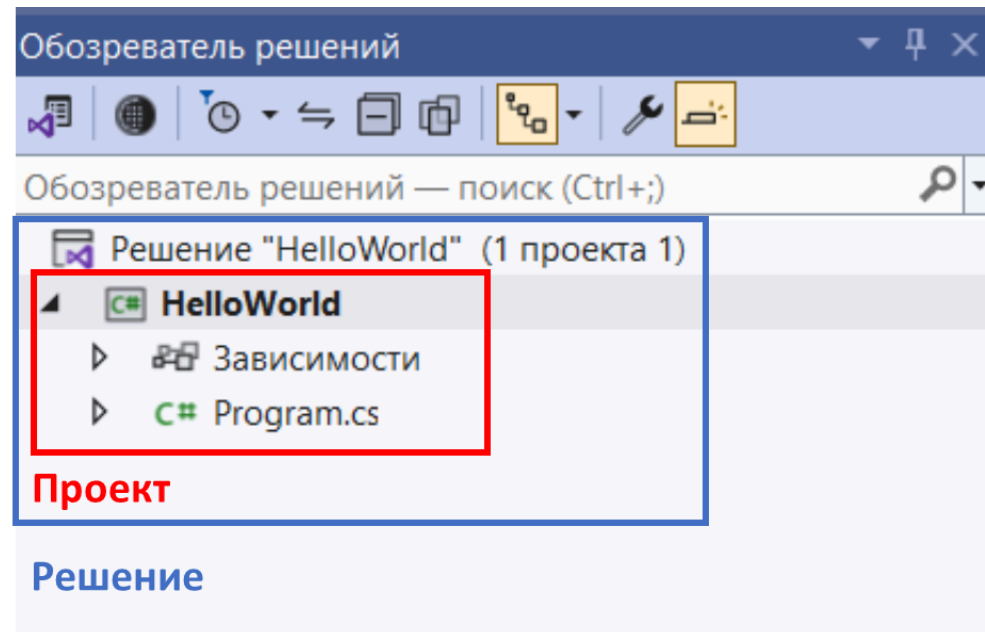
Результат компиляции одного проекта — файл с кодом, готовый к исполнению или подключению к другим проектам.

- **Решение** (`.sln`)

Контейнер для одного или нескольких проектов, объединённых общими зависимостями и задачами. В Visual Studio обычно открывают именно файл решения, но можно работать и с отдельными `.csproj`.

- **Зависимости** (references)

Ссылки внутри проекта на внешние сборки (файлы `.dll` или NuGet-пакеты). Без указания зависимости код из внешней сборки использовать нельзя.



Основные элементы структуры кода в C#:

- **Метод**

Последовательность действий внутри класса. Аналог функции или процедуры в других языках.

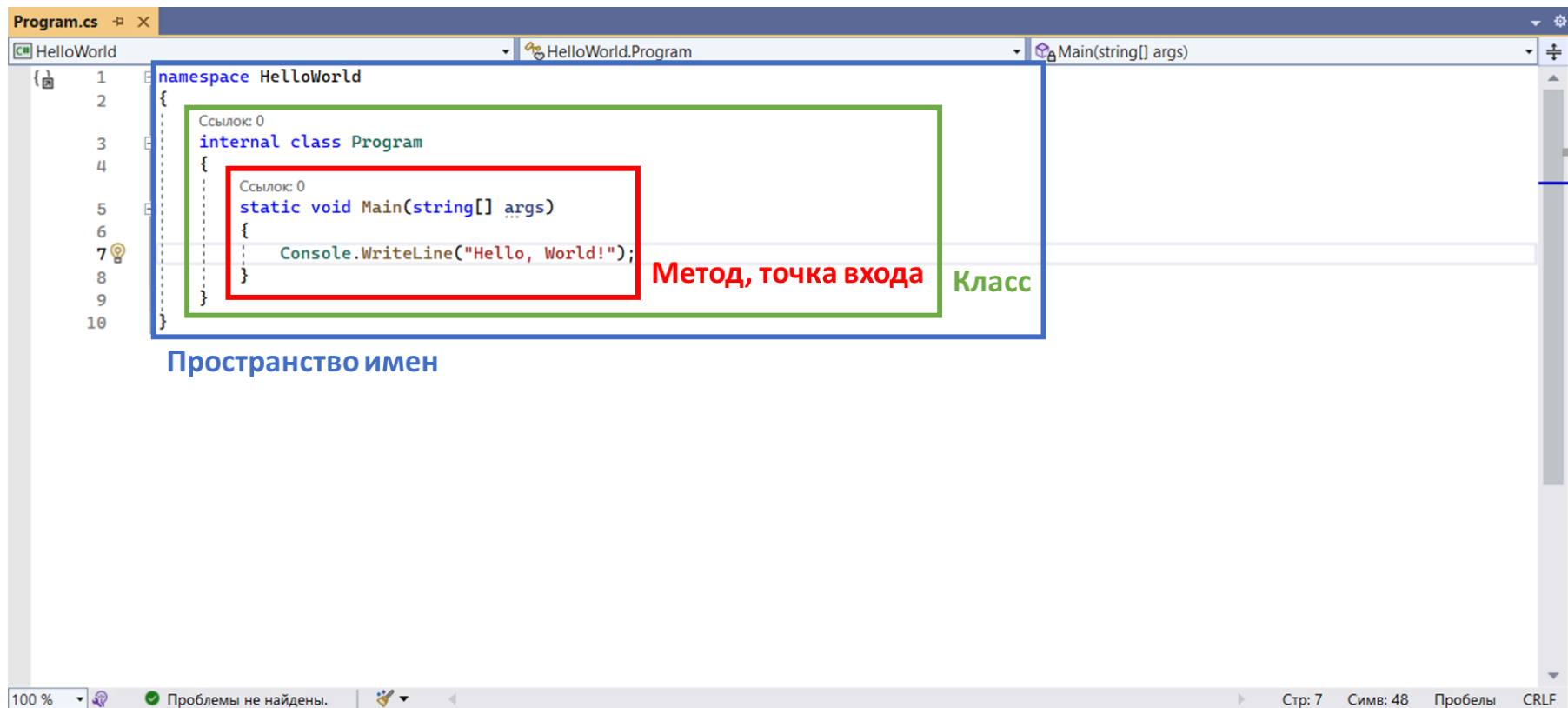
- **Класс**

Логическая группа данных (полей) и методов, объединённых в одну сущность. Любая сборка (.exe или .dll) состоит из множества скомпилированных классов.

- **Пространство имён (namespace)**

Объединение классов и других типов по смыслу и функциональности.

- В одной сборке может храниться несколько пространств имён.
- Классы одного пространства имён могут быть распределены по разным сборкам.



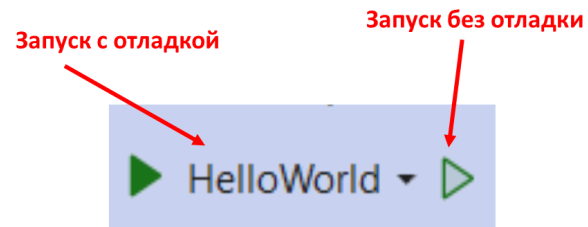
Запуск:

- **Запуск с отладкой**

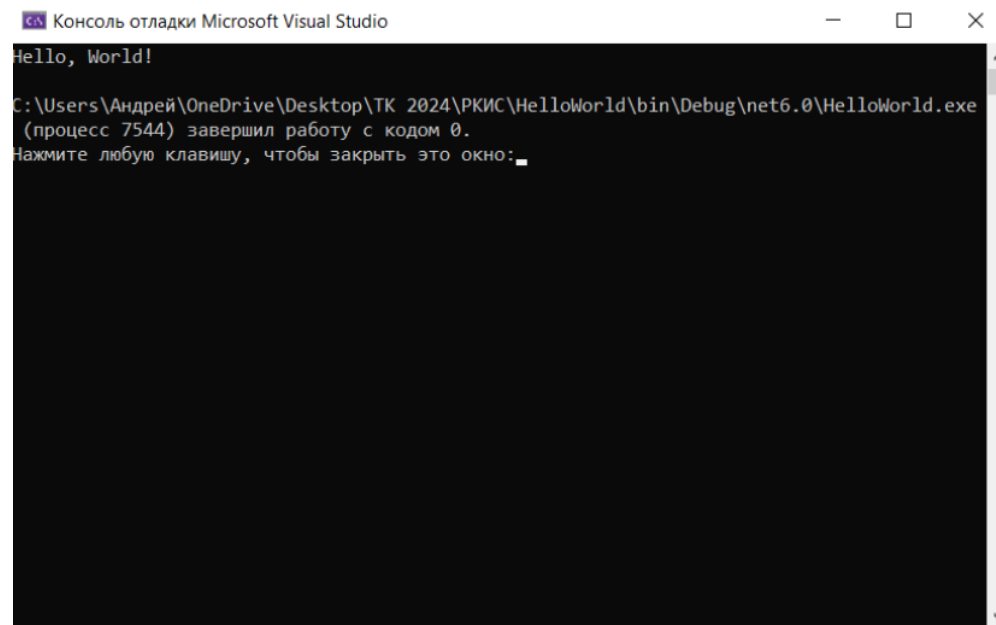
Программа запускается под управлением отладчика — специального инструмента, который позволяет следить за её работой построчно.

- **Запуск без отладки**

Программа запускается как обычное приложение — быстро и без вмешательства отладчика.



После нажатия кнопки *Запуск*, код скомпилируется, после чего приложение запустится. Если нужно скомпилировать код без запуска используйте клавиши `Ctrl+B`.



После успешной компиляции проекта в его корневой папке создаётся поддиректория `bin/Debug`, где лежат готовые сборки (`.exe` и/или `.dll`). Это тот же самый результат, который можно запустить или подключить к другим проектам.

Файлы проекта:

стол > ТК 2024 > РКИС > HelloWorld >

Имя

.vs

bin

obj

HelloWorld.csproj

HelloWorld.sln

Program.cs

Имя

Debug

HelloWorld.csproj.nuget.dgspec.json

HelloWorld.csproj.nuget.g.props

HelloWorld.csproj.nuget.g.targets

project.assets.json

project.nuget.cache

стол > ТК 2024 > РКИС > HelloWorld > bin > Debug > net6.0

Имя

Тип

Размер

HelloWorld.deps.json

HelloWorld.dll

HelloWorld.exe

HelloWorld.pdb

HelloWorld.runtimeconfig.json

Исходный файл J...

Расширение при...

Приложение

Program Debug D...

Исходный файл J...

1 КБ

5 КБ

146 КБ

11 КБ

1 КБ

3 Переменные и типы

Переменная в C# — это именованная область памяти, в которой хранится значение определённого типа. **Под типом понимается множество допустимых значений и операций над ними.** Переменные позволяют работать с данными: сохранять их, изменять и передавать между частями программы. C# — это язык **со статической типизацией**, то есть тип переменной проверяется на этапе компиляции.

Существует два способа объявления переменных:

- **Явное указание типа.**

При явном указании вы сразу записываете тот тип, который нужен

- **Ключевое слово `var`.**

Вместо явного указания типов можно использовать `var`, и компилятор сам выведет нужный тип

Константа — это именованное значение, которое фиксируется при объявлении и не может быть изменено в ходе выполнения программы, обозначается ключевым словом `const`.

С помощью **конверсии типов (`cast`)** можно преобразовывать переменные одного типа в переменные другого типа. Есть два вида преобразований:

- **Неявное преобразование.**

Допускается, когда «младший» тип точно помещается в «старший» без потерь. Можно записать как обычное присваивание.

- **Явное преобразование.**

Нужно, когда возможна потеря данных. В скобках указывается название типа, в который нужно конвертировать значение, например `(int)`.

Числовые типы

```
//Переменная – это именованная область памяти.  
//  
//Тип переменной – это формат области памяти, определяющий множество возможных значений  
// переменной и множество допустимых операций над ней.  
  
int integerNumber;  
// так объявляется переменная: тип (int), затем имя (integerNumber)  
  
// так осуществляется присваивание  
integerNumber = 10;  
  
// double – основной тип чисел с плавающей точкой.  
// Можно совмещать объявление и присваивание.  
double realNumber = 12.34;  
  
// float – тип меньшей точности.  
// Суффикс f говорит, что 1.234 – константа типа float, а не double.  
// Используются в библиотеках работы с графикой в Windows.  
float floatNumber = 1.234f;  
  
//long (большие целые числа). Часто используется для подсчета миллисекунд.  
// L – суффикс констант такого типа, чтобы не перепутать их с int.  
long longIntegerNumber = 30000000000000L;  
  
// Есть и другие типы данных: short, decimal, и т.д.  
// В основном, для чисел вы будете пользоваться int и double, иногда – long и float
```

Тип	Размер (бит)	Диапазон	Пример
byte	8	0 ... 255	byte b = 200;
short	16	−32 768 ... 32 767	short s = −1000;
int	32	−2 147 483 648 ... 2 147 483 647	int i = 50000;
long	64	−9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	long l = 1_000_000_000;
float	32 (плавающая точка)	$\approx \pm 1.5 \times 10^{-45} \dots \pm 3.4 \times 10^{38}$	float f = 3.14f;
double	64	$\approx \pm 5.0 \times 10^{-324} \dots \pm 1.7 \times 10^{308}$	double d = 2.718;
decimal	128 (финансовая точность)	$\pm 1.0 \times 10^{-28} \dots \pm 7.9 \times 10^{28}$ (28–29 знаков)	decimal m = 1000.50m;
sbyte	8	−128 ... 127	sbyte sb = −10
uint	32	0 ... 4 294 967 295	uint ui = 10000
ulong	64	0 ... 18 446 744 073 709 551 615	ulong ul = 100000000000L

Конверсия типов

```
// Конверсия типов (cast) – это преобразование одного типа переменной в другой

int integerNumber = 45;
double doubleNumber = 34.56;

doubleNumber = integerNumber;
// Это неявная конверсия типов: присвоение переменной одного типа
// значения переменной другого типа без дополнительных усилий.
// Она возможна, когда не происходит потери информации

integerNumber = (int)doubleNumber;
// Это явная конверсия типов. В случае, когда конверсия ведет к потере информации
// (в данном случае – дробной части), необходимо явно обозначать свои намерения
// по конверсии.

integerNumber = (int)Math.Round(34.67);
// Округление лучше всего делать не конверсией, а функцией Round.
// Кстати, Math – "математическая библиотека" C# – имеет множество других
// полезных методов.

long longInteger = 40000000000;
integerNumber = (int)longInteger;
// При такой конверсии происходит ошибка переполнения, которая, однако, остается
// незамеченной для компилятора и среды разработки

// Таким образом можно отловить эти ошибки явно
checked
{
    integerNumber = (int)longInteger;
}
```

Строки и символы

```
//Строки – это последовательности символов
string myString = "Hello, world!";

// + – это операция "приписывания" одной строки к другой:
string s = "Hello" + " " + "world";

// Можно обращаться к отдельным символам
char c = myString[1]; //'e' – нумерация символов с нуля.
char myChar = 'e'; // одинарные кавычки используются для символов. Двойные – для строк.

//У строк есть собственные методы и переменные (правильно называть это свойствами),
//которые позволяют узнать информацию о строке
Console.WriteLine(myString.Length);

myString = myString.Substring(0, 5);
Console.WriteLine(myString);

string strangeSymbols = "© 2014 Σμβόλο";

//Тип string может иметь особое значение – null.
//Это не пустая строка, а отсутствие всякой строки.
myString = null;

//Интересно, что тип int такого значения иметь не может.
//int a=null;

int number = int.Parse("42"); //Из строки в число
string numString = 42.ToString(); // Из числа в строку
double number2 = double.Parse("34.42"); // Зависит от настроек операционной системы

//Следующий вызов не зависит от настроек и всегда ожидает точку в качестве разделителя:
number2 = double.Parse("34.42", CultureInfo.InvariantCulture);

//Следующий вызов не зависит от настроек и всегда использует точку в качестве разделителя:
string invariantNumber2 = number2.ToString(CultureInfo.InvariantCulture);
Console.WriteLine(invariantNumber2); //34.42
```

Базовые операции

```
int a = 23;
int b = 45;
double angle = 1.4;

// Математические операции записываются естественным образом
int c = (a + b) / 2;

//Класс Math содержит полезные методы и константы
Console.WriteLine(Math.Sin(angle));

var d = a - b;
/* часто понятно, какого типа должна быть переменная. В этом случае можно писать var
 * Компилятор самостоятельно догадается, что именно вы имели в виду
 */

// это целое число
var e = a / 2;
// это число с плавающей точкой
var f = a / 2.0;

c = b = a;
/* Как это работает? b=a – оператор присвоения, но он имеет собственное
 * возвращаемое значение (равное a)
 * Поэтому c = b = a выполняется так:
 * – b присваивается a
 * – c присваивается результату b=a, который также равен a
 * В итоге все три переменные будут равны
 */
```

```
a -= 4;
// То же самое, что a=a-4, аналогично с другими операциями.

a++;
//Оператор инкремента
//То же самое, что a=a+1

a--;
//Оператор декремента
//То же самое, что a=a-1

++a;
//То же самое, что a=a+1, но с одним отличием:

a = 5;
Console.WriteLine(a++);
// выведет 5

a = 5;
Console.WriteLine(++a);
// выведет 6
```

4 Ошибки

Виды ошибок

- **Ошибки на этапе компиляции (compile-time errors)**

Нарушения синтаксиса, отсутствующие или неверные типы. Компилятор или IDE подскажут, где исправить.

- **Ошибки во время исполнения (runtime errors / exceptions)**

Возникают при выполнении программы (деление на ноль, `NullPointerException`, `IndexOutOfRangeException` и т.д.).

- **Стилистические ошибки**

Не влияют на выполнение, но затрудняют чтение и сопровождение кода. Важны для командной работы.



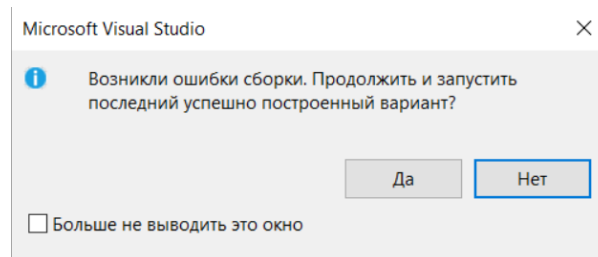
Ошибки на этапе компиляции

Как выглядят и что делать

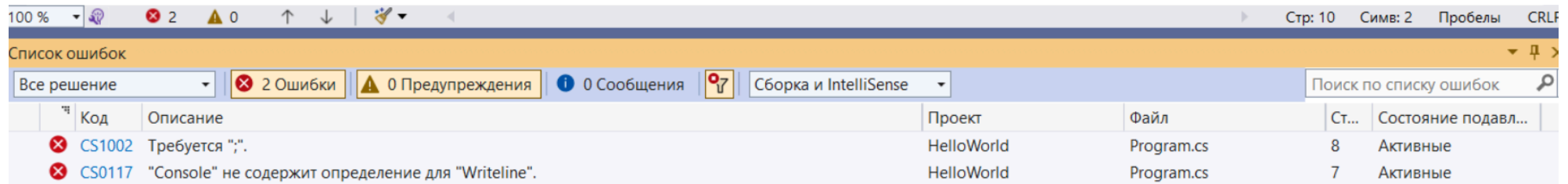
- IDE (Visual Studio / VS Code) подчёркивает синтаксические ошибки (красным) и показывает список ошибок.

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

- Если вы запустите программу появится предупреждение



- В списке ошибок указано: файл, номер строки, сообщение компилятора — прочитайте его.



- Исправьте код

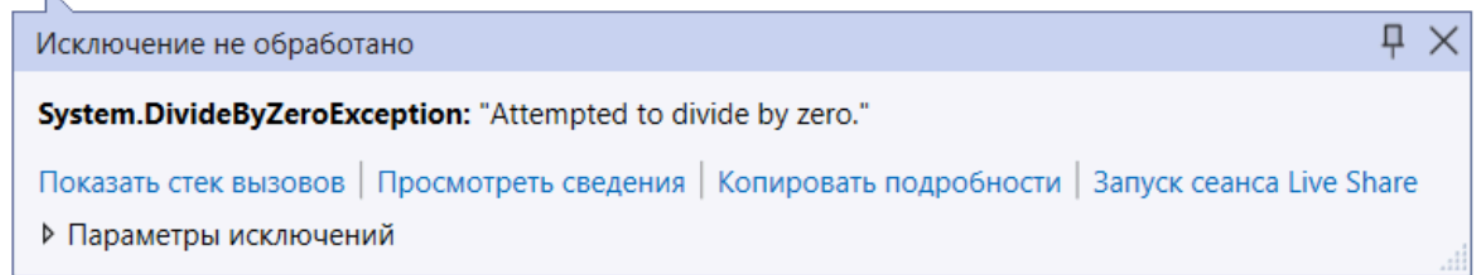
```
static void Main(string[] args)
{
    Console.WriteLine("Hello, world!");
}
```

Ошибки во время исполнения (exceptions)

Что это такое

- Это ошибки, которые не отлавливаются на этапе компиляции, их так же называют runtime-ошибками
- В случае возникновения возвращается объект исключения (exception) со всей необходимой информацией
- Рассмотрим на примере ошибки деления на ноль (запустим с отладкой)

```
static void Main(string[] args)
{
    int a = 0;
    Console.WriteLine(1/a);
}
```



Отладка (debugging)

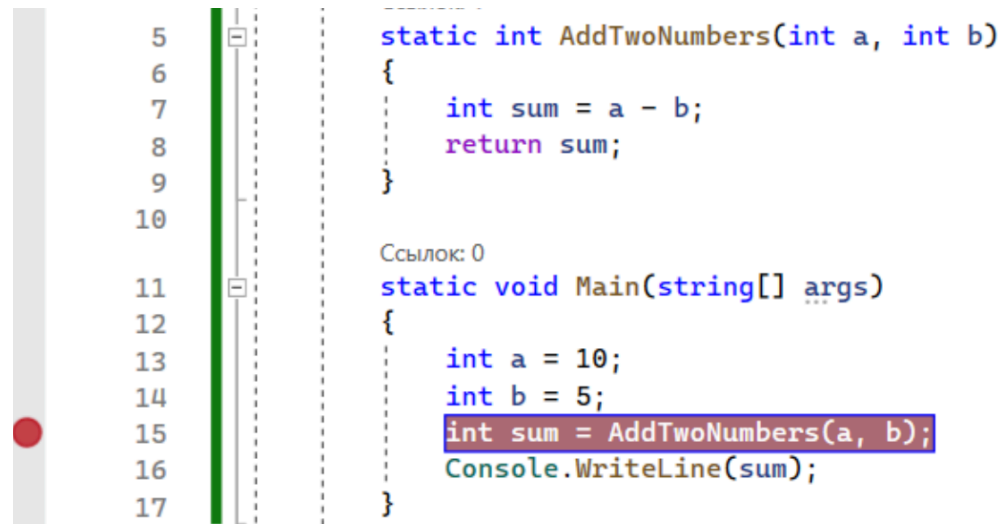
Зачем отлаживать

- Позволяет пошагово пройти программу, посмотреть значения переменных, стек вызовов и понять, где логика работает неправильно.

Рассмотрим на примере. Код возвращает неверное значение и мы хотим понять, где находится ошибка, с помощью отладки.

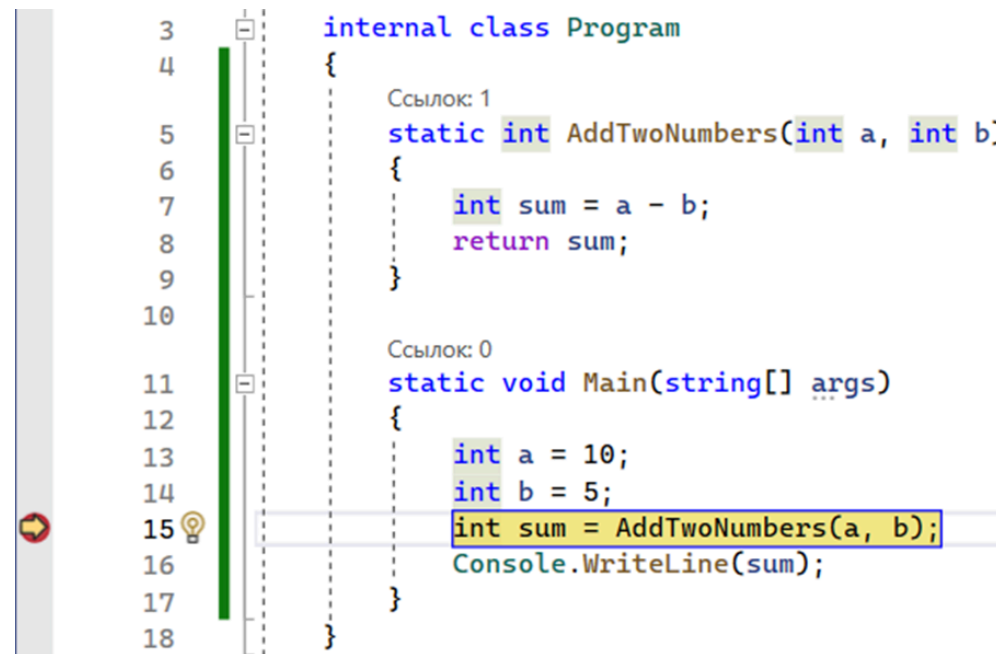
Базовый сценарий

1. Поставьте **точку останова** (breakpoint) слева от строки кода.

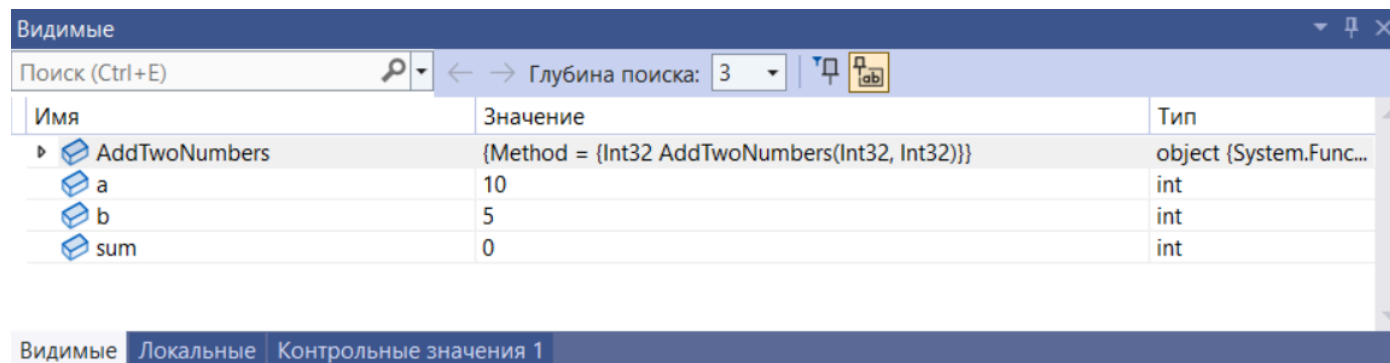


2. Запустите проект в режиме Debug.

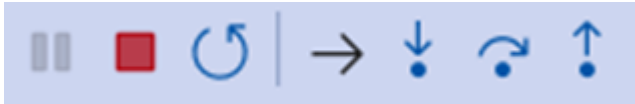
3. Когда выполнение остановится — наведите курсор на переменную, чтобы увидеть её значение.



4. Так же можно посмотреть значение всех видимых переменных в специальном окне.



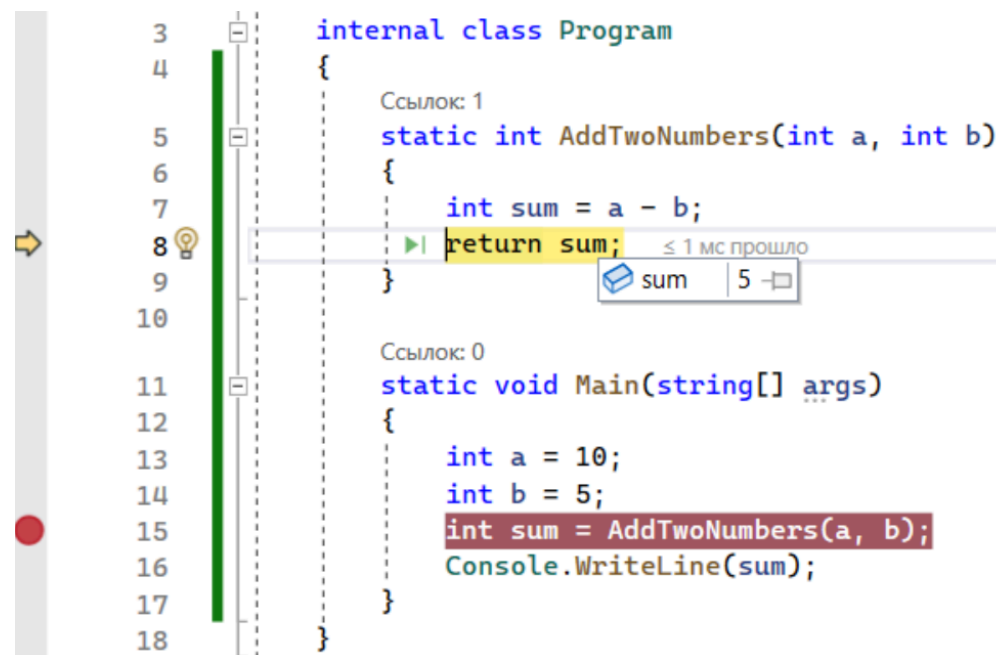
Для перемещения по коду во время отладки, либо его остановки можно использовать кнопки в верхнем меню либо горячие клавиши.



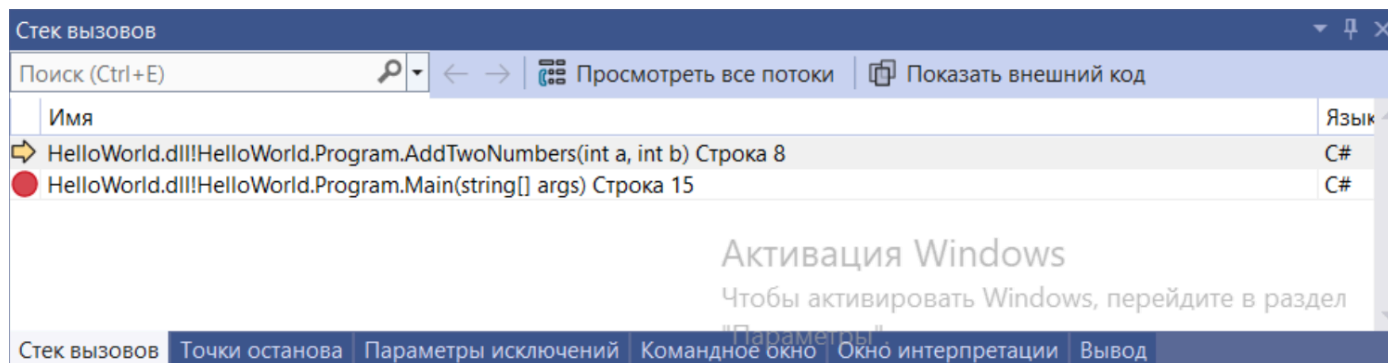
Горячие клавиши

- F5 — запустить/продолжить (Start/Continue)
- F10 — Step Over (выполнить строку, не заходя в методы)
- F11 — Step Into (заход в метод)
- Shift+F11 — Step Out (выйти из метода)
- Ctrl+Shift+F9 / убрать все брейкпоинты (Visual Studio)

5. Зайдем внутрь метода AddTwoNumbers, найдем ошибку и закончим отладку



6. Также во время отладки отображается стек вызовов методов, с помощью него можно увидеть последовательность их вызовов, а также переходить на различные этапы выполнения программы.



Стилистические ошибки (code style)

- Единый стиль облегчает чтение и уменьшает количество багов при правках.
- Особенно критично для командных проектов и крупных приложений.

Рекомендованные ресурсы

- [Microsoft C# coding conventions](#)

Короткие правила

- Имена должны быть **осмысленными** и на **английском**.
- Классы и методы: **PascalCase** (например `AddTwoNumbers` , `UserRepository`).
- Переменные, параметры, поля: **camelCase** (например `userName` , `totalSum`).
- Константы: использовать для значений, которые не будут меняться `PascalCase` .
- Имена не должны быть слишком короткими или содержать артикли (`the` , `a` , `an`).
- Методы должны содержать глагол в названии (`CalculateTotal` , `SaveUser`).
- Используйте `var` там, где тип очевиден по правой части; иначе указывайте явный тип для читаемости.

Ссылки

- [Скачать Microsoft Visual Studio](#)
 - [Скачать JetBrains Rider](#)
 - [Учебник по языку C# от Microsoft](#)
 - [Учебник по языку C# от Metanit](#)
 - [Как работать с Git, через Visual Studio](#)
 - [Microsoft C# coding conventions](#)
-

Практическое задание

Что нужно сделать:

1. Выберите себе напарника, с которым вы будете работать.
2. **Сделайте форк** моего репозитория на GitHub (ссылка будет выдана преподавателем).
3. **Добавьте своего напарника** в раздел *Collaborators* своего форка:
 - Откройте свой форк → `Settings` → `Collaborators` → добавьте GitHub-ник партнёра.
4. **Склонируйте репозиторий** себе на компьютер с помощью Git.
5. **Создайте консольное приложение** в Visual Studio или Rider, назовите его `ToDoList`.
6. **Создайте файл `.gitignore`**, добавьте туда папки `bin`, `obj`, `.vs`.
7. **Измените текст**, который выводится в консоль (*например: работу выполнили Иванов и Петров*).
8. **Напишите программу**:
 - Попросите пользователя ввести имя, фамилию и год рождения
 - Получите введенные пользователем значения (`Console.ReadLine`) и запишите их в соответствующие переменные
 - Переведите год рождения в целое число и вычтите текущий год, чтобы получить возраст
 - Вывести сообщение вида: `Добавлен пользователь <Имя> <Фамилия>, возраст - <возраст>`
9. **Сделайте коммит и push** изменений.
10. **Создайте README-файл** и кратко пишите, что вы сделали.
11. **Создайте Pull Request** в исходный репозиторий:
 - Назовите его строго по шаблону:
Фамилия1 Фамилия2 номер_группы
(*например: Иванов Петров 3831*)