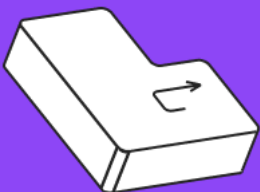




Начало работы: .Net и Visual Studio

Углубленный курс C#

Урок 1



Оглавление

Введение	3
На этом уроке	4
Платформа .Net	5
Сборка мусора	8
Сборка проекта	9
Инструменты разработки	12
Виды приложений	14
Начало работы с Visual Studio	16
Создание нового проекта	16
Знакомство с главным окном Visual Studio	19
Структура проекта	22
Работа с кодом	24
Запуск и отладка приложения	27
Стиль написания кода	35
Заключение	37
Термины, используемые в лекции	38
Домашнее задание	39
Используемая литература	39
Ссылка на гитхаб	40

Введение

Приветствую всех на нашем курсе “Углубленный C#”, в ходе изучения которого мы научимся профессионально писать код на языке C#.

Этот курс научит вас применять основные инструменты языка и пользоваться библиотекой платформы, писать аккуратный и красивый код. В наши дни самым популярным направлением разработки с использованием .Net является backend-разработка в проектах web-приложений.

В нашей и в зарубежных странах разработчики ПО, в том числе, C#-разработчики являются крайне востребованной специальностью. Рынок труда постоянно испытывает нехватку специалистов. Частота появления новых ИТ-компаний не оставляет шанса остаться без работы даже новичку в этой области.

Давайте представим, что вам нужно разработать приложение интернет-магазина, чат-бота для Telegram или же утилиту для поиска файлов на диске вашего компьютера. C# позволяет реализовать любую из этих задач.



C# является современным, динамично развивающимся языком программирования. Язык вобрал в себя все лучшее от своих предшественников, таких как C++ и Java.

Язык обладает огромным потенциалом и широко применяется в разработке desktop, web и игровых приложений. Он входит в тройку лидеров по разработке Android-игр (главным образом, благодаря платформе Unity).

Хочется сказать пару слов о грейдах разработчиков.

Новичком или же младшим разработчиком, в терминологии отрасли это Junior, считается разработчик, только закончивший учебу. На этом этапе карьеры вам будут помогать и курировать вашу деятельность более опытные разработчики из команды. На первый взгляд, задачи могут показаться скучными и рутинными. Действительно, скорее всего, поначалу вам будут передавать в разработку какие-то базовые элементы, но вместе с ними вы получите бесценный опыт, с которым перейдете на уровень выше.

Спустя полгода-год работы, ваша квалификация вырастет и вы перейдете в статус Middle-разработчика. С новым статусом вы получите больше автономии, меньше контроля. Однако по серьезным решениям все еще придется искать совета.

Senior-разработчик - это не только знания, но и ответственность как за себя, так и за всех участников команды рангом ниже. Сложно дать однозначный ответ, сколько

потребуется времени чтобы им стать - переход от мидла к сеньору сильно зависит как от способностей и личных качеств самого разработчика, его желания и возможности учиться, так и от проектов, над которыми ему пришлось поработать.

На этом уроке

Цель урока: понять, как устроена платформа .Net, как работают ее основные компоненты и какие инструменты мы будем применять в работе над проектами.

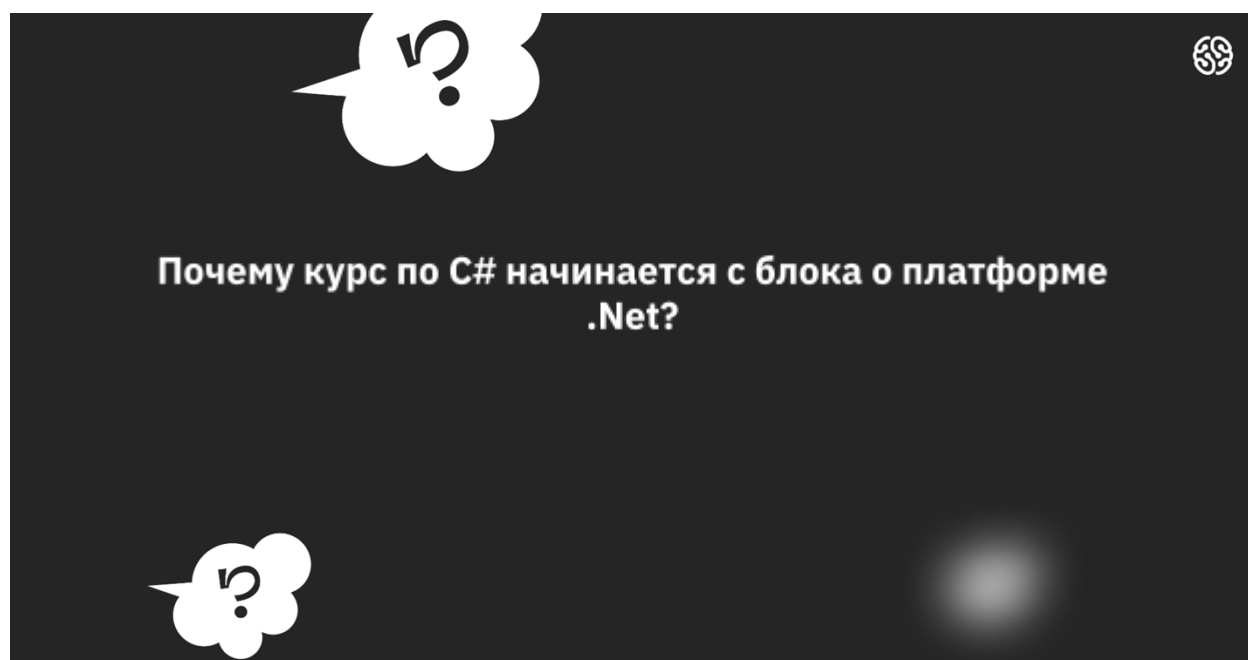
Как достигнем цели:

- разберем устройство платформы .Net и .Net framework, их отличия, а также что такое управляемый и неуправляемый код
- изучим инструменты Visual Studio, используемые для создания и отладки приложений
- выясним, какие виды приложений существуют и для чего предназначен каждый из них
- изучим структуру .Net-проекта и назначение отдельных его файлов
- напишем наше первое приложение при помощи Visual Studio
- научимся использовать отладчик
- узнаем про стиль написания кода



Для эффективного освоения материала курса “Углубленный C#” лучше всего освежить в памяти содержание материалов подготовительных курсов: **введение в программирование, основы C#, базы данных.**

Платформа .Net



Ответ: Язык C# - это язык разработки под программную среду .Net. Программа, написанная на этом языке, может работать там, где установлен .Net. Таким образом, целесообразно начинать погружаться в обучение именно с информации о .Net.

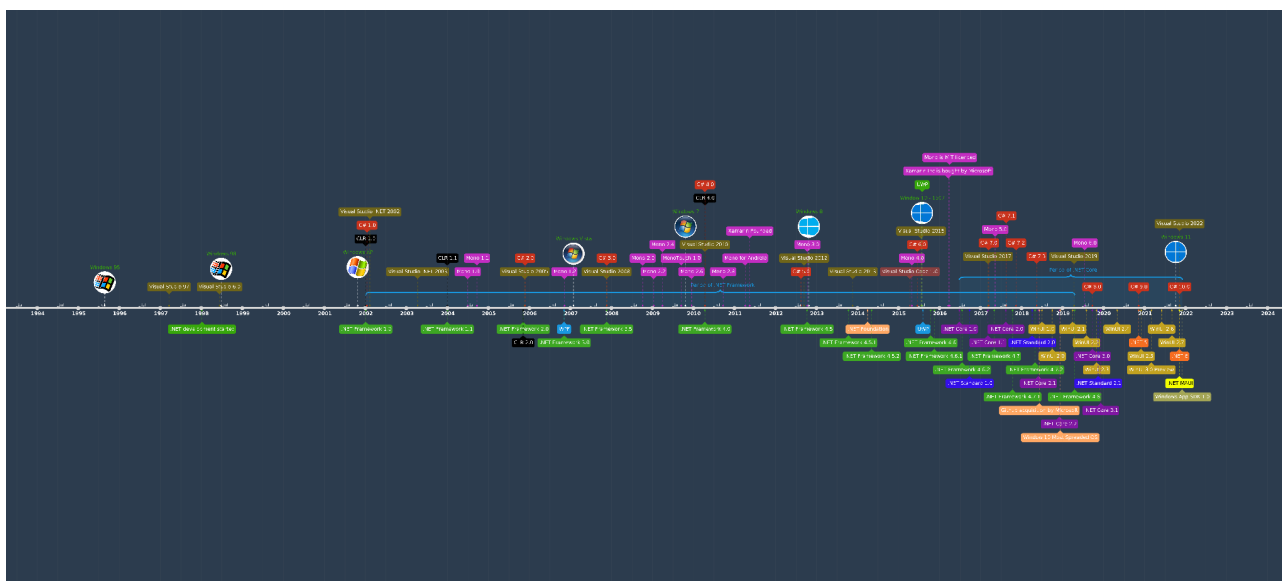
Возможно, кому-то из вас, кто уже начал интересоваться подробностями разработки на языке C#, встречались термины “.Net”, “.Net Framework”, “.Net Core”.

Если не встречались или вы только приступаете к знакомству с языком, то первым делом мы проведем параллель и разграничим функционал этих платформ.

Еще в далеком 1999 году корпорация Microsoft, будучи лидером рынка операционных систем, страдала от отсутствия единого стандарта разработки под свою платформу. К тому же в то время росла популярность платформы Java. Все это поспособствовало тому, что корпорация Microsoft приступила к созданию своей единой платформы, призванной упростить разработку и эксплуатацию приложений. Первый релиз такой платформы, названной .Net Framework, состоялся в 2002 году.

За последующие десятилетия платформа получила признание и была высоко оценена разработчиками Windows. Текущая версия платформы 4.8.1 была выпущена в 2022 году (*информация актуальна на январь 2023 года*).

Параллельно с этим в 2016 году Microsoft выпускает совершенно новую версию платформы под названием .Net Core. В отличие от .Net Framework, работающей исключительно под Windows, .Net Core является мультиплатформенной, поддерживая Windows, Linux, Mac OS, мобильные платформы. Помимо всего прочего исходный код платформы является [открытым](#) (open source). Начиная с пятой версии платформы, .Net Core была переименована в .Net. Текущая версия фреймворка - 7, выпущена в 2022 году.



День рождения .Net Framework так же можно считать и днем рождения языка C#, первое появление которого было в составе Visual Studio .Net в 2002 году. С самого начала язык позиционировался как основной язык разработки под платформу. По ходу роста популярности платформы .Net росла и популярность C#. Он активно развивается, и по сей день де-факто являясь наиболее популярным языком разработки под платформу.

Текущая версия C# - 11. Она поддерживается .Net 7.0 и средой разработки Visual Studio 2022. Наряду с C# платформа .Net поддерживает и другие языки, такие как: F#, Visual Basic .Net, C++/CLI.

💡 Язык программирования C# был разработан исключительно как язык разработки для .Net, тогда как другие языки программирования, например, VB.Net, C++/CLI были лишь адаптированы для разработки под эту платформу.

Важно знать, что .Net Framework 4.8.1 полноценно поддерживает только лишь C# 7 версии. Поэтому для целей обучения мы будем писать проекты исключительно на .Net (бывший .Net Core).

Подробнее о преимуществах .Net.

В первую очередь, .Net - это технология, предоставляющая разработчику единую среду для разработки ПО. Неважно, на каком языке пишет разработчик, будь то C#, C++/CLI или, например, F#: у него есть доступ к библиотеке классов платформы .Net - [FCL](#) (Framework classes library), предоставляющей обширные возможности для реализации его задач.

Библиотека включает в себя классы для создания консольных, графических и Web-приложений, параллельного выполнения кода, сетевого взаимодействия и многое другое.

Вторая важная составляющая .Net - это единая среда для выполнения кода, которая называется [CLR](#). Это расшифровывается как Common Language Runtime - общая среда выполнения приложений, написанных под платформу .Net.

[CLR](#) является прослойкой между приложением и операционной системой. Такой подход гарантирует, что приложение сможет запуститься на любой операционной системе, где установлена версия платформы .Net, под которую оно было разработано.

💡 Иногда приложение все-таки приходится адаптировать под конкретную операционную систему. Например, при работе с файлами в операционных системах Windows, Linux и MacOS используются разные разделители при указании пути к файлу (“~/documents/file.txt”, “C:\Documents\file.txt”).

Ключевые особенности [CLR](#):

- умеет оптимизировать “на лету” приложение под архитектуру конкретной платформы;

- включает в свой состав “сборщик мусора” - механизм, упрощающий работу с памятью для программистов путем автоматического освобождения памяти переменных, вышедших из зоны видимости;
- гарантирует “безопасное выполнение” кода, полученного из сторонних источников: платформа поддерживает строгую типизацию и безопасный доступ к памяти;
- предоставляет различные службы диагностики, отладки, получения дампов памяти, трассировки и наблюдаемости выполняемого кода (Совокупность этих служб обеспечивает гибкий мониторинг приложений. Тут есть все инструменты позволяющие оптимизировать выполнение вашего кода под конкретную платформу, а также настроить его гибкое взаимодействие с компонентами системы);
- умеет эффективно взаимодействовать с “[неуправляемым кодом](#)”.



Код, согласно терминологии .Net, выполняемый под управлением [CLR](#), называется [управляемым кодом](#). И, наоборот, код, выполняемый вне [CLR](#), называется [неуправляемым кодом](#).

Сборка мусора

Одним из важнейших компонентов [CLR](#) является [GC](#) (Garbage collector).

Сборщик мусора является инструментом автоматического освобождения памяти в .Net, благодаря которому программисту не приходится заниматься этим вручную.

Для каждого объекта, созданного приложением, [CLR](#) выделяет память из управляемой кучи (managed heap).



Куча (Heap) - это структура данных, используемая для выделения памяти приложению.

Пока в управляемой куче есть свободные адреса памяти, [CLR](#) резервирует их под новые объекты. Чтобы этот процесс продолжался, необходимо очищать память, высвобождая ее из-под неиспользуемых объектов.

Процесс сборки мусора работает параллельно выполнению программы и никак не влияет на ее ход.

Когда сборщик мусора выполняет свою работу, он проверяет наличие неиспользуемых приложением объектов в управляемой куче, а затем выполняет необходимые операции, чтобы освободить память.

В процессе работы, сборщик мусора разделяет все созданные объекты на 3 поколения:

- нулевое поколение содержит объекты, которые еще не подвергались анализу на предмет возможности их удаления из памяти;
- первое поколение содержит объекты, которые не были собраны при сборке мусора нулевого поколения;
- второе поколение содержит объекты, прошедшие более одной сборки мусора.

Частота проверки объектов зависит от поколения, к которому они относятся. Чем старше поколение, тем реже происходит его проверка.

Кроме этих трех поколений в управляемой куче есть специальная область, называемая LOH (Large object heap), предназначенная для размещения объектов, которые занимают большой объем памяти. Эта область проверяется с той же частотой, что и второе поколение. Иногда LOH называют 3 поколением сборщика мусора.

Сборка проекта



Термин “сборка” означает процесс компиляции и последующей компоновки его результатов в приложение или библиотеку.

[Исходный код программы](#) - это код, написанный человеком и понятный человеку. Он предназначен для описания логики работы программы.

[CLR](#) не умеет выполнять такой код. Более того, машинный анализ исходного кода - задача трудоемкая и если бы он выполнялся каждый раз при запуске приложения это было бы слишком медленно.

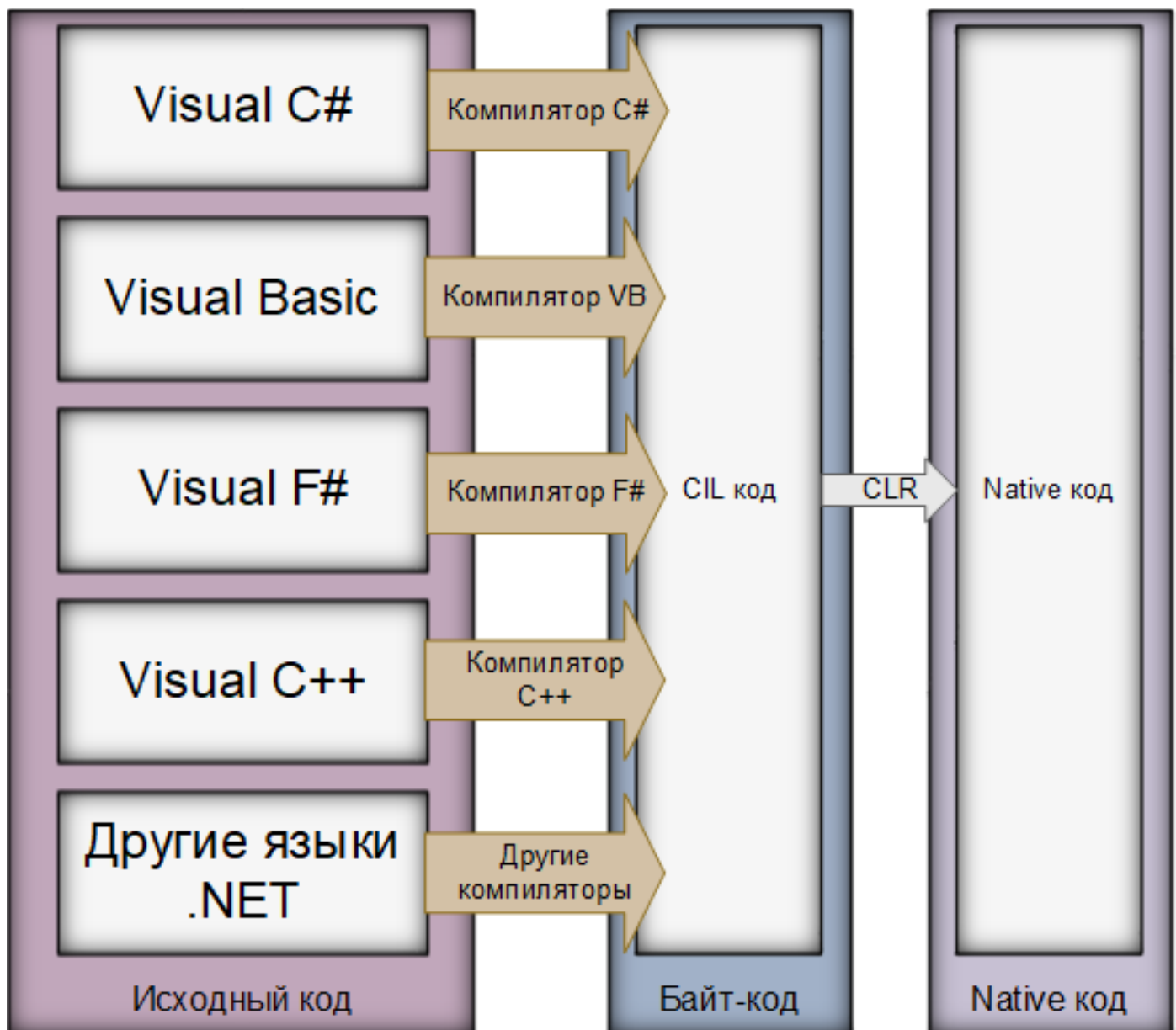
Код, который умеет исполнять [CLR](#), называется Intermediate Language или сокращенно [IL](#).

Любое приложение созданное для .Net, по сути, является [IL](#)-кодом, хранящимся на диске компьютера или его памяти.

Чтобы преобразовать [исходный код](#) в код [IL](#), необходима специальная программа - компилятор. Это приложение, которое переводит исходный код в код [IL](#).

Каждый язык программирования, предназначенный для .Net, имеет свой собственный компилятор. Компилятор C# входит в среду разработки Visual Studio.

Ранее в этой лекции упоминалось, что [CLR](#) является прослойкой между приложением и операционной системой. Процесс выполнения приложения включает в себя преобразование кода [IL](#) в нативные коды (машинные коды), понятные ОС, а также совместимые с текущей архитектурой процессора. [CLR](#) делает такое преобразование “на лету”. За эту работу отвечает компонент, называющийся [JIT](#) (Just in time compiler).



Приведенная иллюстрация показывает путь приложения от [исходного кода](#) до его запуска на целевой платформе.

Итак, подытожим, глядя на иллюстрацию:

- [ИСХОДНЫЙ КОД](#) - это текст программы, написанной разработчиком;

- компилятор - это специальная программа, преобразующая [исходный код](#) в [IL](#) (результатом работы компилятора является исполняемый файл или же библиотека);
- [IL](#) (CIL на иллюстрации) - это Intermediate Language, специальный байт-код, понятный [CLR](#);
- [CLR](#) - Common language runtime, общая среда выполнения приложений .Net;
- native код - код, понятный целевой платформе (Windows, Linux, Mac).

Инструменты разработки

Основным инструментом разработки под платформу .Net является Microsoft Visual Studio. Именно его мы будем использовать на протяжении всего нашего курса.

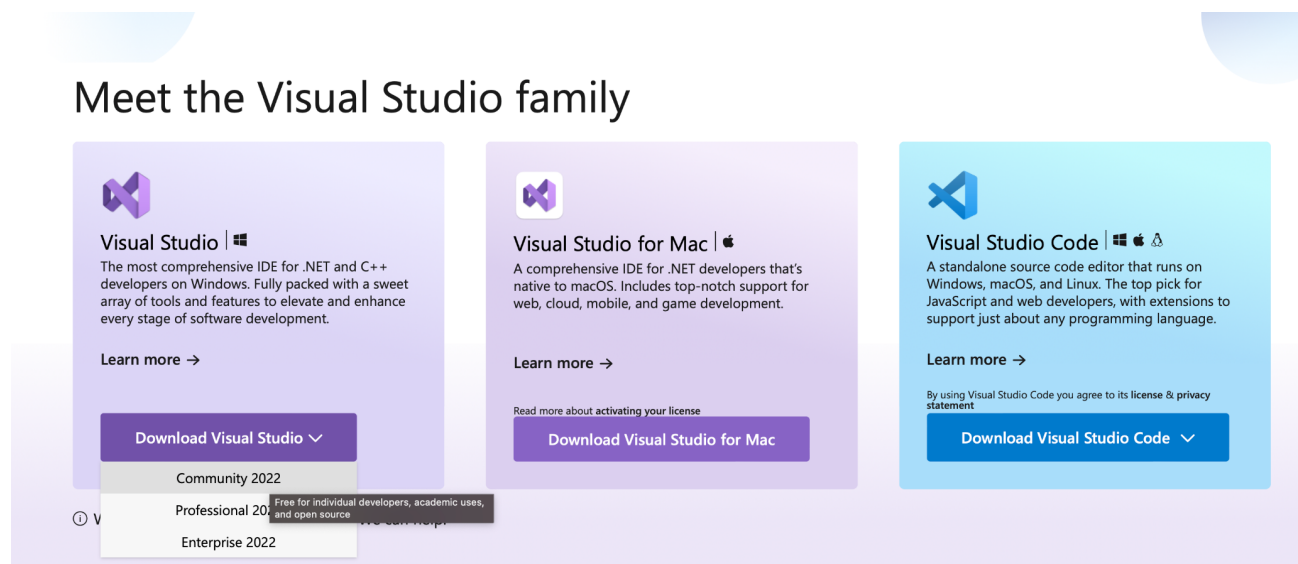
Visual Studio - это интегрированная среда разработки приложений (integrated development environment или IDE). Это специальная программа, поддерживающая написание, компиляцию (сборку), отладку кода и публикацию готовых приложений. Редактор кода Visual Studio обладает множеством функций, которые пригодятся как новичкам, так и более опытным программистам для улучшения процесса разработки.

Еще одна важная особенность Visual Studio - это ее цена. Для индивидуальных разработчиков она бесплатна.

На сегодняшний день актуальная версия называется Visual Studio 2022 и она может быть свободно загружена по ссылке: <https://visualstudio.microsoft.com/>.

Visual Studio существует в нескольких редакциях, доступных под операционные системы Windows и Mac OS.

Версия, которую мы будем использовать, называется Community 2022.

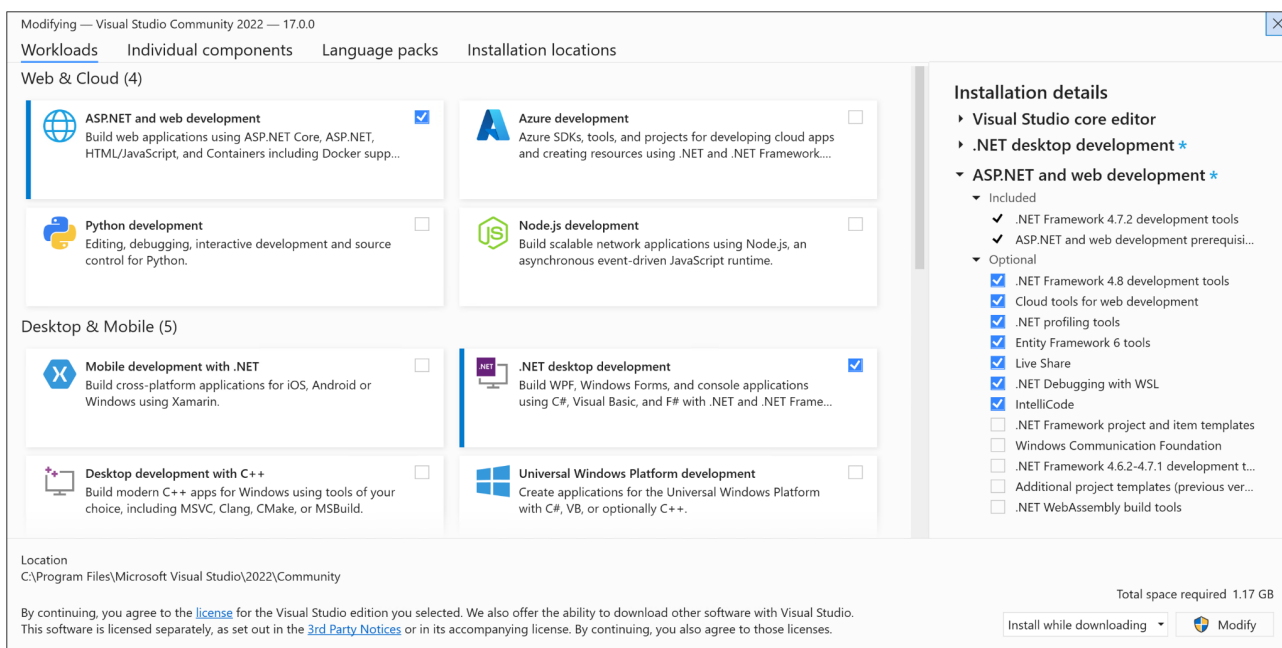


Многие путают Visual Studio с Visual Studio Code, считая, что это разные версии одного и того же приложения. Но это не так. Visual Studio **Code** - является продвинутым редактором кода и не подходит для целей нашего курса в отличие от Visual Studio.

После запуска установщика Visual Studio необходимо выбрать опции среды, которые будут установлены.

Для нашего курса следует установить (отметить галочками) следующие опции:

- ASP.Net & web development - разработка web-приложений;
- .Net desktop development. - разработка настольных приложений или сервисов.



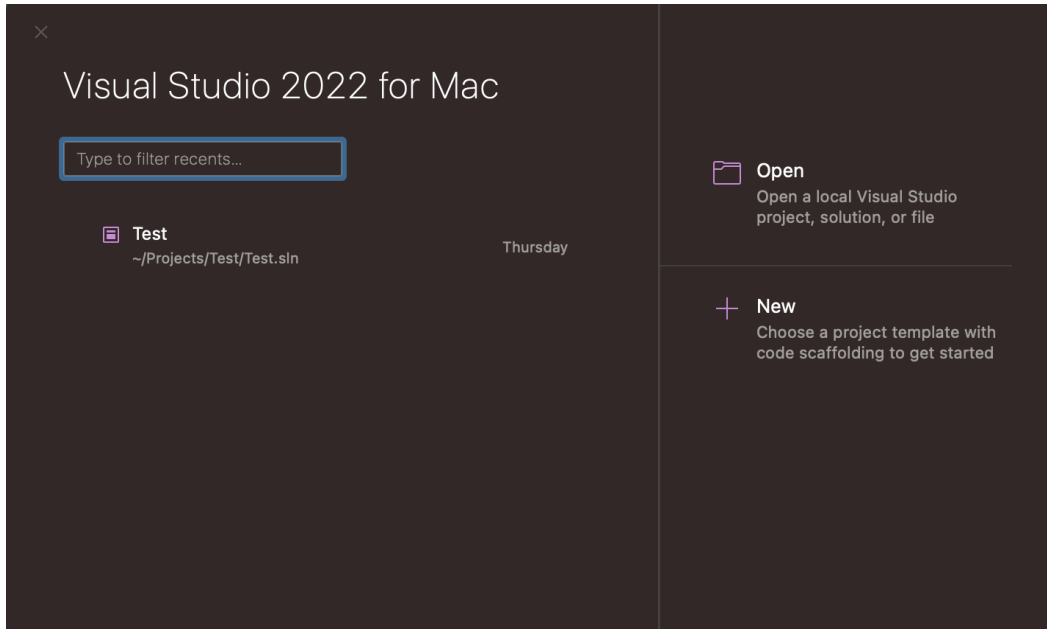
После установки иконка Visual Studio появится в меню приложений и/или на рабочем столе.



Если в ходе работы вы поймете, что чего-то не хватает, то вы всегда сможете повторно запустить установщик и установить недостающее.

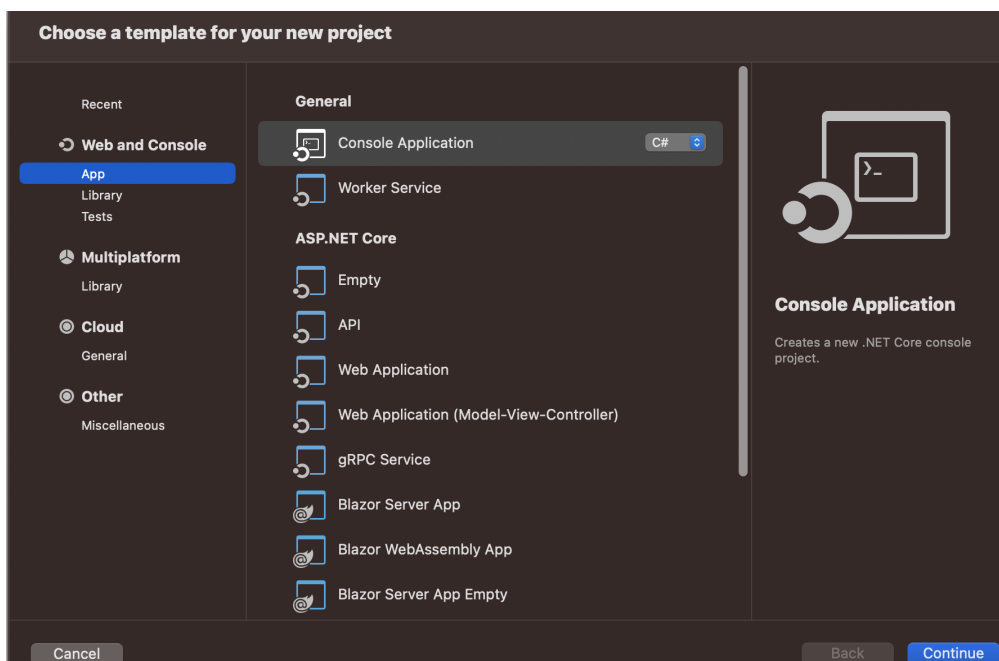
Виды приложений

После запуска Visual Studio, среда предложит создать новый проект или же продолжить работу над уже существующим.



Рассмотрим создание нового проекта.

После нажатия **New** откроется окно выбора шаблона проекта.



Шаблоны проектов удобно структурированы в группы:

- Web and Console - группа, позволяющая выбрать шаблоны проектов для web и desktop разработки

- App - приложения
- Library - библиотеки классов
- Tests - проекты unit-тестирования
- Multiplatform - шаблоны кроссплатформенных проектов
 - Library - библиотеки классов
- Cloud - облачные приложения
 - General - проекты для работы с облачной платформой Microsoft Azure
- Other - прочее
 - Miscellaneous - все то, что не вошло в предыдущие группы (если при установке вы выбрали рекомендованные выше опции, то в этой подгруппе будет находиться только проект пустого [решения](#))



В рамках программы мы будем работать со всеми основными типами проектов, находящимися в группах Web and Console.

Теперь подробнее о шаблонах.

В группе App находятся шаблоны приложений.

General->Console Application - это консольное приложение. Такие приложения запускаются из терминала (cmd.exe) или PowerShell, а взаимодействие с пользователем происходит посредством текстового интерфейса. Пример консольного приложения - это утилита ping, позволяющая проверить доступность произвольного узла в локальной или глобальной сети.

General->Worker service / General->Windows service (в версии для Windows) - приложение, не имеющее интерфейса, основным предназначением которого является фоновое выполнение различных операций. Например, это может быть сервис синхронизации вашего локального каталога с облачным хранилищем (Google drive).

ASP.Net (ASP.Net Core) - шаблоны веб приложений (мы научимся работать с ними во второй половине нашего курса).

Начало работы с Visual Studio

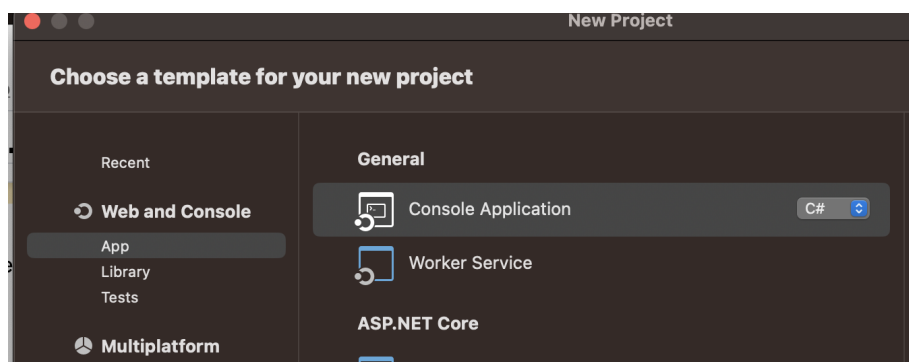
Создание нового проекта

Попробуем создать наше первое приложение. Пускай это будет приложение, считывающее при запуске ваше имя и печатающее приветствие в ответ.

Для изучения основ языка лучше всего подходит такой тип приложения как консольное.

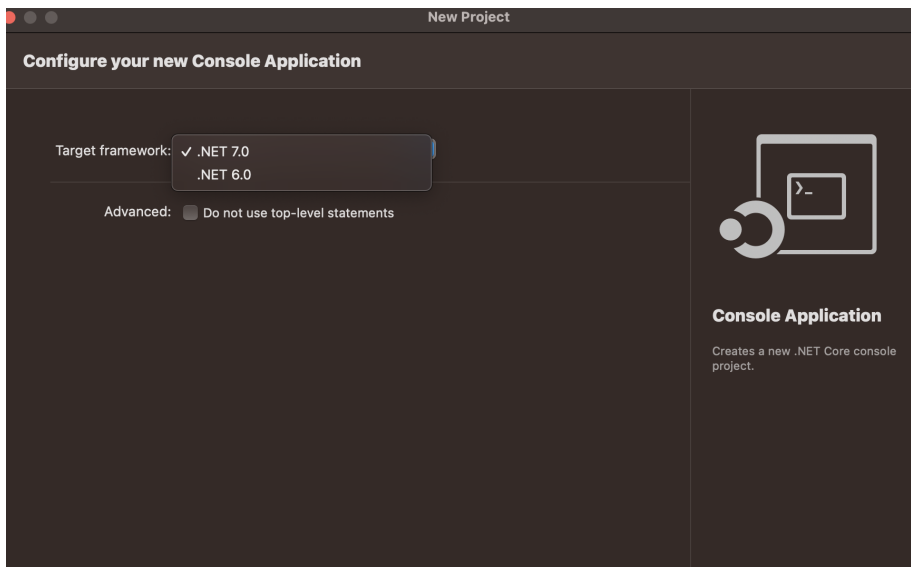
Давайте разберемся, какие действия нужно совершить, чтобы создать свое приложение.

В меню выбора шаблона проекта нужно выбрать Console Application и нажать **Continue**.



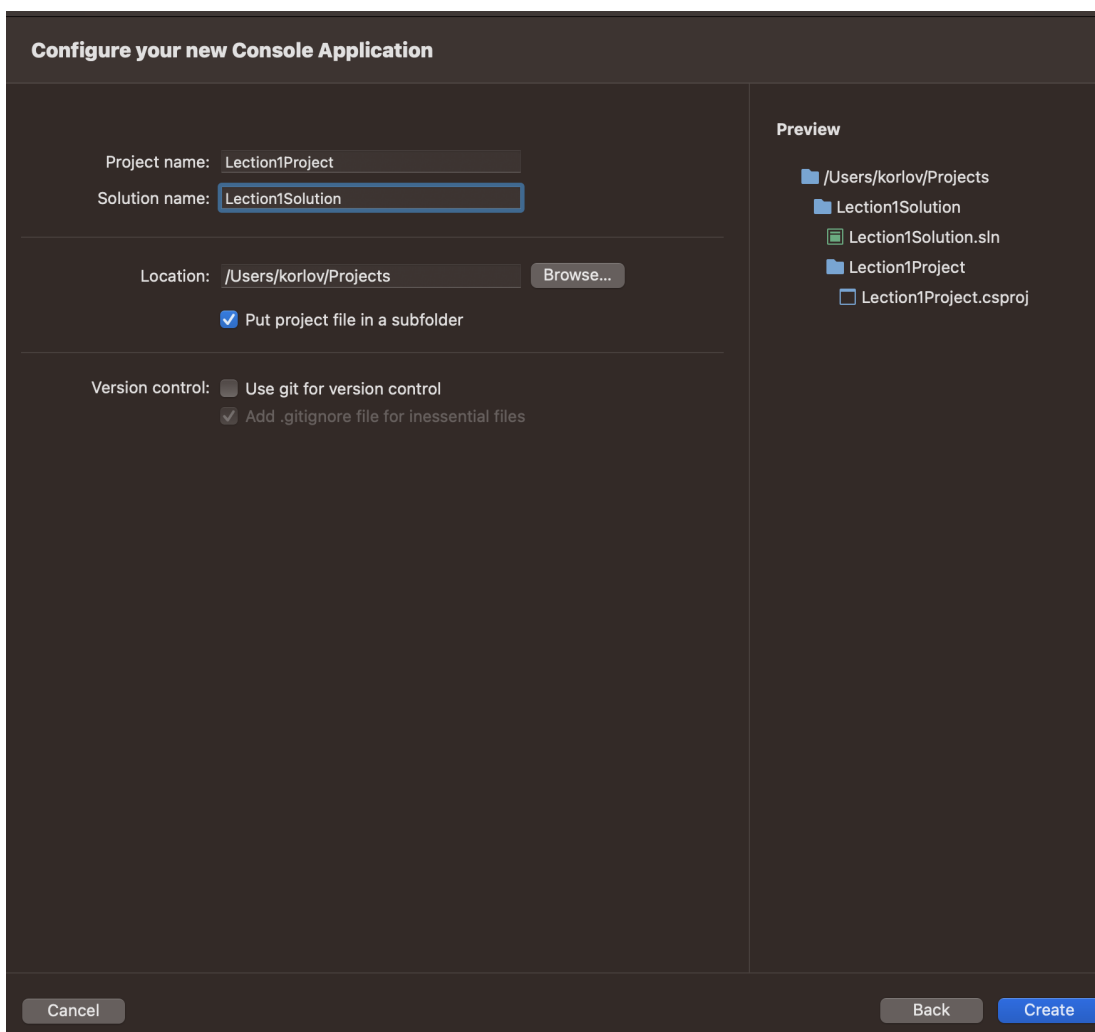
В следующем окне будет предложено выбрать версию .Net, которую будет поддерживать наше приложение. Выберите версию и нажмите **Continue**.

Выбор чекбокса “Do not use top-level statements” означает, что код шаблона приложения будет включать в себя инструкции его вызова, иными словами - функцию Main.



В следующем окне выберите название проекта (Project name) и [решения](#) (Solution name), а также путь, по которому будет сохранен проект (Location).

Чекбокс после Version control указывает на то, что вы планируете использовать систему контроля версий в ходе работы над проектом.



После нажатия кнопки **Create** откроется основное окно среды разработки.

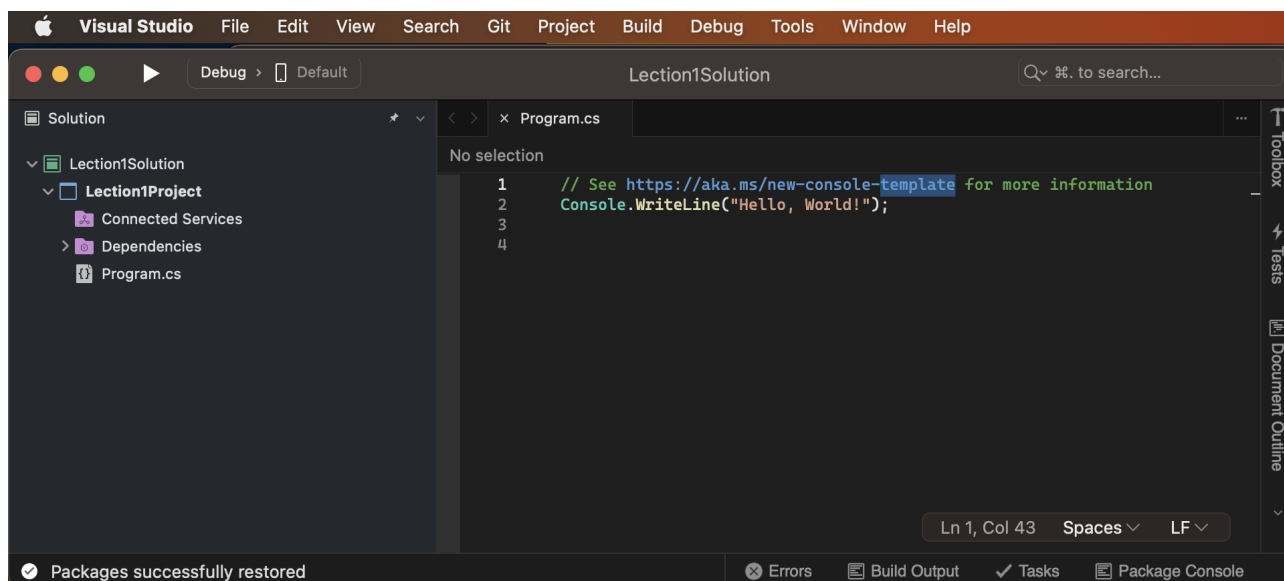


Также консольный проект можно создать из терминала, выполнив команду в предварительно созданном каталоге:

dotnet new console

Знакомство с главным окном Visual Studio

Давайте разберемся с возможностями, которые нам предоставляет среда разработки.



Главное окно Visual Studio (на картинке выше) разделено на две части:

- редактор кода - отображает содержимое текущего файла;
- список файлов проекта в виде дерева - отображает структуру каталога проекта и некоторые его настройки.

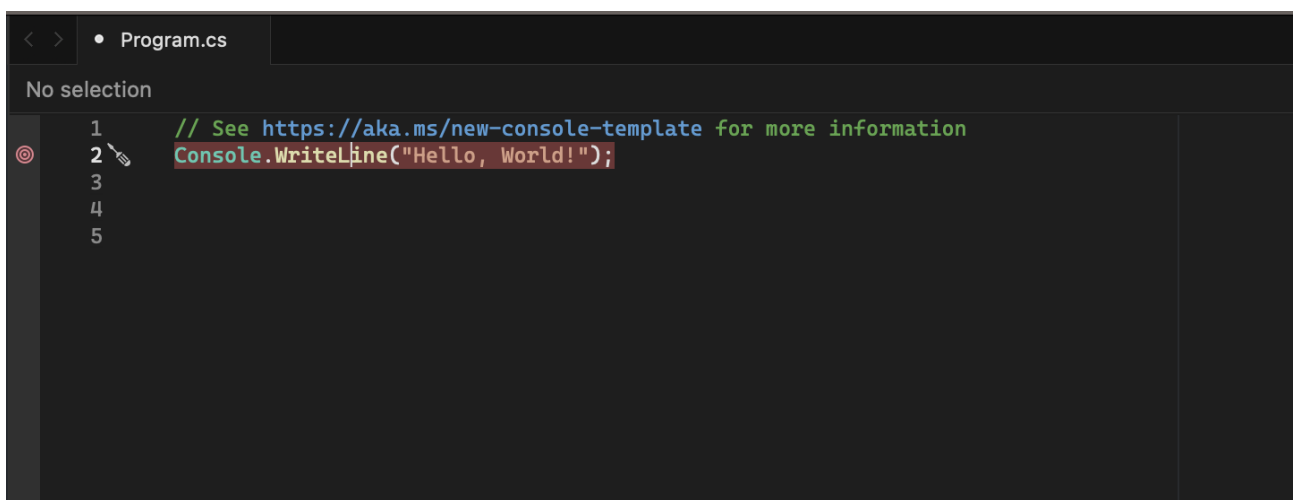
Также в верхней части экрана находится меню из следующих групп:

- Visual Studio - в подменю этой группы можно настроить среду разработки;
- File - работа с файлами: здесь вы можете сохранить, закрыть или начать новый проект;
- Edit - управление вводом текста (копировать, вырезать, вставить);
- View - позволяет настроить отображаемые окна среды разработки;
- Search - поиск по файлам/текущему файлу;
- Git - работа с системой управления версиями (если она подключена к проекту);
- Project - позволяет добавлять элементы в проект, а также его настраивать;
- Build - меню сборки проекта, содержит следующие подпункты:
 - Build - собирает проект/решение: иными словами, генерирует IL-код из исходного кода вашей программы (если решение уже собрано и вы не изменили ни строки кода, то сборка не происходит)

- Rebuild - начинает новую сборку проекта/[решения](#), предварительно удалив результаты предыдущей сборки
- Clean all - удаляет результаты сборки проекта/[решения](#)
- [Debug](#) - меню отладки, содержит следующие подпункты:
 - Windows - отображает инструменты отладки:
 - Breakpoints - работа с точками останова;
 - Locals - значения локальных переменных;
 - Watch - здесь вы можете добавить произвольную переменную, значение которой будет отображаться в ходе отладочной сессии;
 - Immediate - редактор, позволяющий написать и выполнить произвольный кусок кода в ходе выполнения программы;
 - Call stack - отображение стека вызовов текущей функции;
 - Threads - отображение параллельных потоков;
 - Live visual tree - интерактивное отображение элементов графического интерфейса в процессе отладки приложения;
 - Modules - отображение модулей, из которых состоит программа (библиотеки);
 - Start debugging - запускает процесс отладки
 - Start without debugging - выполняет приложение без возможностей отладки
 - Run with - позволяет настроить различные параметры запуска приложения (например, переменные окружения или рабочий каталог)
 - Stop - останавливает процесс выполнения приложения
 - Debug application - запускает процесс отладки произвольного приложения
 - Attach to process - позволяет присоединиться и начать отладку, запущенного вне среды Visual Studio приложения
 - Dettach - действие обратное Attach: прекращает отладку приложения, запущенного вне среды
 - Step into - переход на уровень ниже по стеку вызовов

- Step over - переход к следующей строке кода
- Step out - переход на уровень выше по стеку вызовов
- Show next statement - отображение следующей выполняемой инструкции
- Toggle breakpoint - включение/выключение точки останова на текущей строке кода
- New breakpoint - подменю, позволяет производить расширенное управление точками останова
- View breakpoints - отображение всех точек останова
- Disable breakpoint - отключение точки останова под курсором
- Disable all breakpoints - отключение всех точек останова
- Remove all breakpoints - удаление всех точек останова
- Show disassembly - показывает текущие выполняемые машинные коды
- Run unit tests - запускает на выполнение [ЮНИТ-ТЕСТЫ](#)
- Tools - вспомогательные инструменты Visual Studio;
- Window - управление расположением окон;
- Help - различные виды справки.

Большую часть пространства экрана занимает редактор кода.

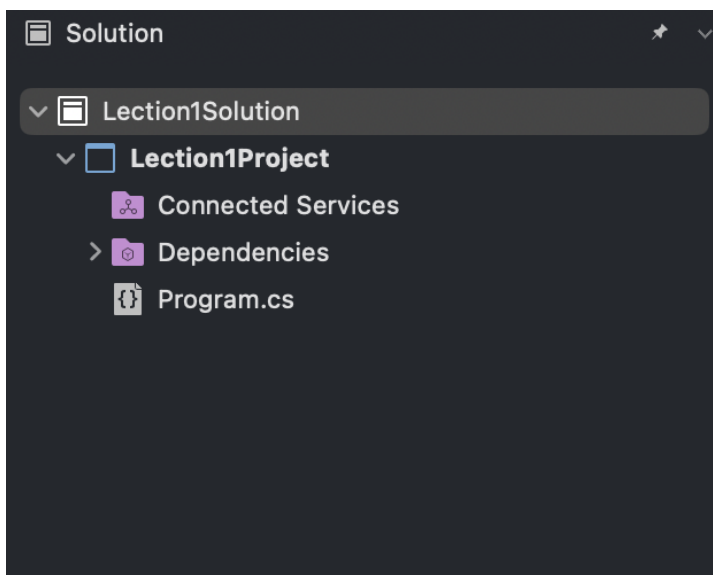


В заголовке редактора содержится имя файла (Program.cs), который открыт на редактирование в данный момент. Строки в редакторе кода пронумерованы.

Нумерация помогает ориентироваться в коде при совместной работе над проектом: например, вы можете попросить коллегу обратить внимание на код на определенной строке. Левее нумерации, есть небольшая вертикальная панель, на которой можно видеть текущие точки останова программы (красный кружок на 2 строке).

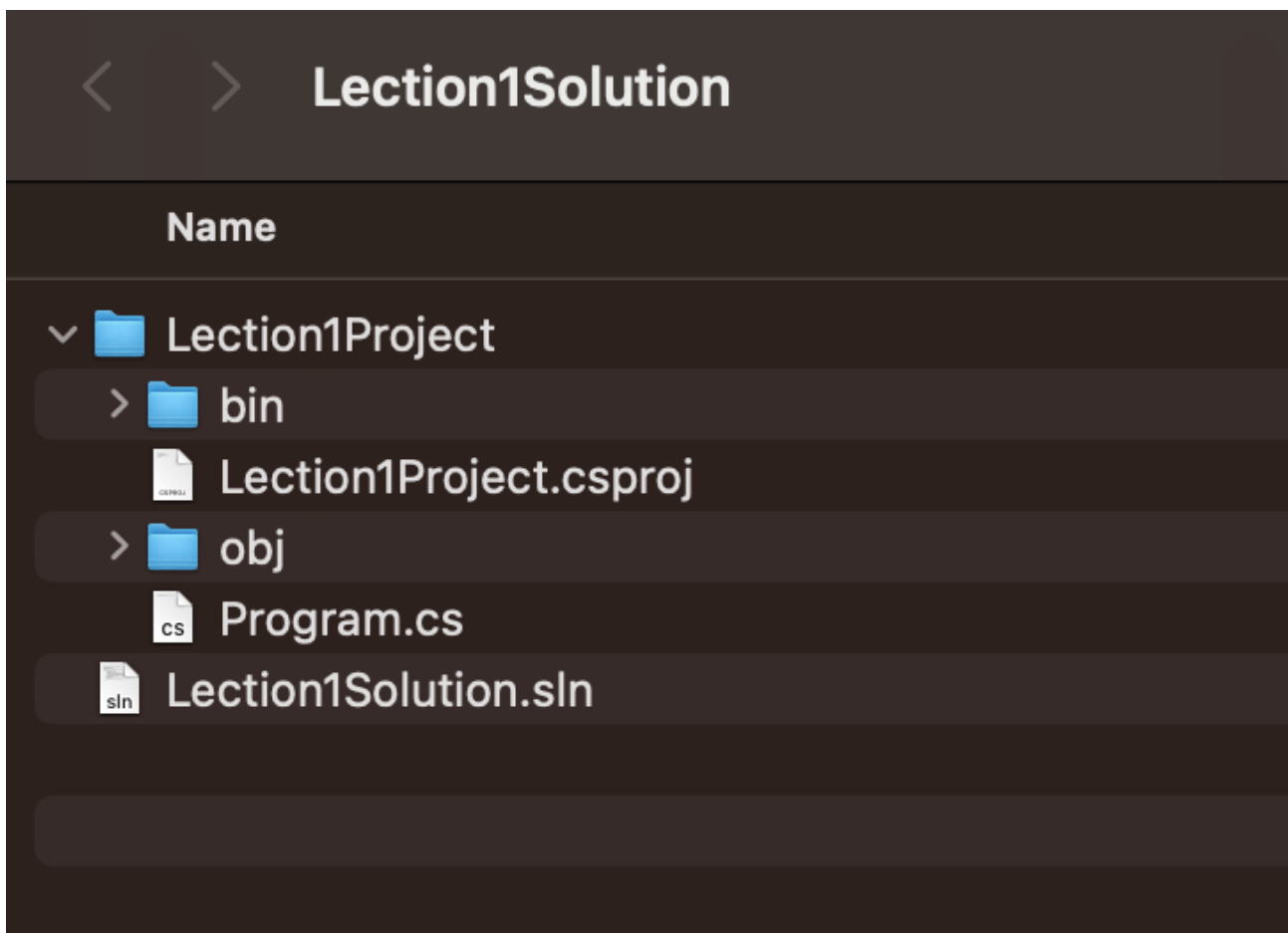
Структура проекта

Левее (а у кого-то правее) редактора кода находится окно, отображающее структуру текущего [решения](#).



💡 Проекты в Visual Studio объединены в [решение](#) (Solution) - логическая сущность, которая может содержать один или несколько проектов.

Открыв папку с [решением](#) в файловом менеджере, мы увидим следующее:



Каталог [решения](#) (Lecture1Solution на иллюстрации):

- Lecture1Project - каталог проекта, входящего в [решение](#)
 - bin - каталог с результатами сборки проекта
 - Lecture1Project.csproj - файл с настройками проекта
 - obj - объекты, образованные в ходе сборки проекта
 - Program.cs - [исходный код](#) проекта
- Lecture1Solution.sln - файл с настройками [решения](#)



В зависимости от выбранного имени проекта и имени [решения](#) название файлов и папок может быть иным.

Программа не обязательно должна состоять из одного исходного файла. Большие проекты могут содержать сотни, а то и тысячи файлов с [исходным кодом](#).

Исходные файлы могут содержать пространства имен, которые в свою очередь содержат классы, структуры, интерфейсы, перечисления, делегаты и другие пространства имен. Все эти понятия мы разберем в ходе работы над программой курса.

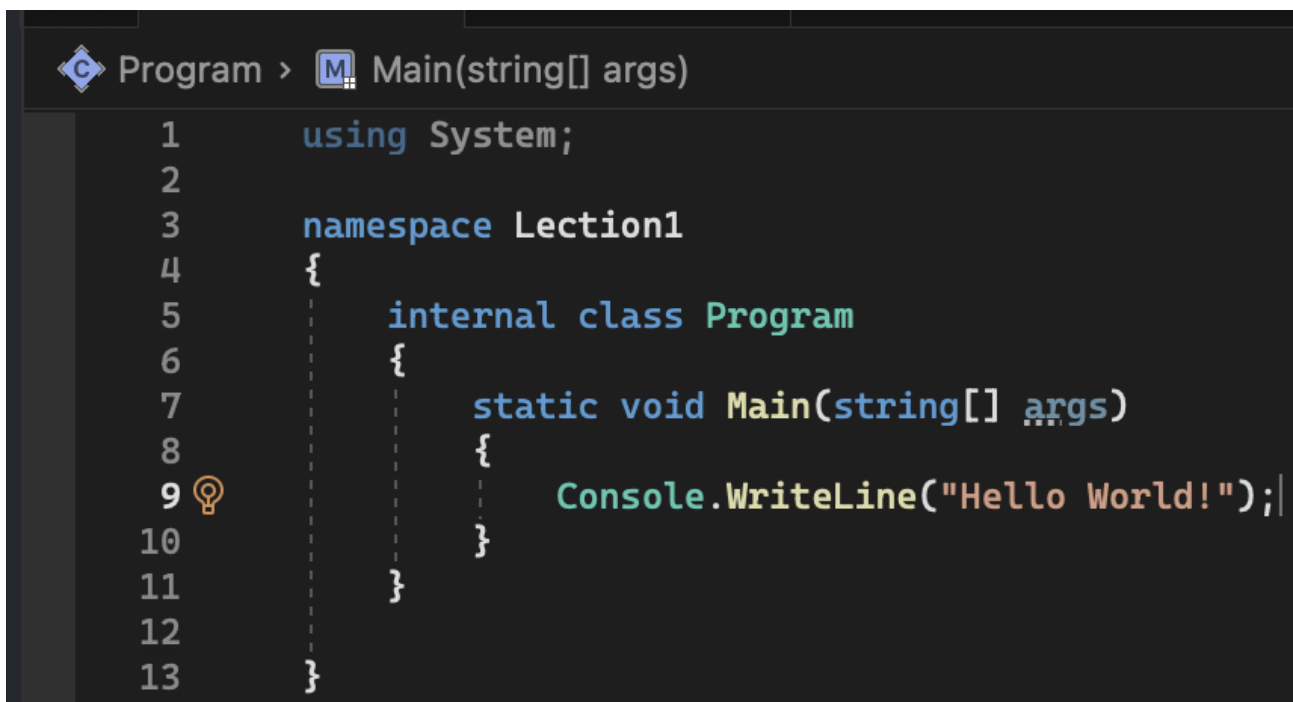
Работа с кодом

Вернемся к коду нашего приложения. Проект создан в версии Visual Studio 2022 с поддержкой 7.0 версии фреймворка.

Код, созданный автоматически, лаконичен и фактически содержит одну команду вывода на консоль строки: “Hello, World!”.

```
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
4
5
```

В предыдущих версиях Visual Studio без поддержки 6.0 и в более поздних версиях фреймворка этот код выглядел следующим образом:



```
Program > Main(string[] args)
1      using System;
2
3      namespace Lektion1
4      {
5          internal class Program
6          {
7              static void Main(string[] args)
8              {
9                  Console.WriteLine("Hello World!");
10             }
11         }
12     }
13 }
```

Причина в том, что последние версии .Net (начиная с 6.0) создают упрощенный шаблон консольного приложения и компилятор C# в момент сборки проекта “дописывает” за вас недостающие инструкции.

Для удобства изучения C# рекомендуется на первых порах самостоятельно дописывать недостающие инструкции. Это поможет запомнить основные элементы программы или же просто отметить чекбокс “Do not use top-level statements” на этапе создания проекта.



Также консольный проект с включенными инструкциями верхнего уровня можно создать используя следующую команду:

```
dotnet new console --use-program-main
```

Разберем подробнее, из чего состоит наше автоматически сгенерированное приложение.

Строка 1 содержит директиву **using System** - это указание компилятору подключить к нашему проекту в ходе его сборки пространство имен System. Это позволит нам использовать классы из этого пространства имен без обязательного указания пространства имен перед именем (последние версии C# не требуют указания данного пространства имен, автоматически добавляя его на этапе компиляции).

Строка 9 содержит обращение к методу **WriteLine** статического класса **Console**.

Если бы в строке 1 мы не добавили директиву **using System**, то строка 9 в предыдущих версиях C# выглядела бы следующим образом (последние версии C# автоматически добавляют пространство имен System в код программы при его сборке, именно поэтому на скриншоте **System** окрашен в серый цвет что означает избыточность его указания):

```
8      {  
9      System.Console.WriteLine("Hello World!");  
10     }
```

Казалось бы никакой разницы тут нет, однако в случае если к классу Console мы обращаемся более одного раза, код становится менее лаконичным из-за множественных обращений к одному и тому же пространству имен.

Кстати, в строке 3 мы уже сами задаем пространство имен к которому будет относиться наше приложение.

5 строка объявляет класс нашего приложения - **internal class Program**.

Тут сразу два ключевых слова:

- **internal** - это модификатор доступа, указывающий на область видимости класса Program,
- **class** - это ключевое слово, объявляющее класс,

за которыми следует имя класса - Program.

7 строка содержит объявление функции - **static void Main(string[] args)**.

Ключевые слова:

- `static` - указывает на то, что функцию нужно вызывать, не создавая экземпляра класса,
- `void` - указывает на тип возвращаемого функцией значения (в случае с `void` функция ничего не возвращает).

Имя функции **Main**, за ним следуют скобки, в которых указаны параметры, передаваемые в функцию - **`string[] args`** - массив строк с именем `args`.

При запуске консольного приложения CLR будет искать именно эту функцию, чтобы начать с нее выполнение программы. Впрочем имя функции и класса ее содержащего можно переопределить в настройках проекта.

Давайте изменим наше приложение таким образом, чтобы оно, запускаясь из командной строки, получало ваше имя в виде параметра и использовало его для приветствия. В случае, если число параметров не будет равняться 1, программа будет выводить уже знакомую нам надпись "Hello, World!".

```
No selection
1      namespace Lektion1
2      {
3          internal class Program
4          {
5              static void Main(string[] args)
6              {
7                  if (args.Length == 1)
8                  {
9                      string name = args[0];
10                     string message = $"Привет {name}";
11                     Console.WriteLine(message);
12                 }
13                 else
14                 {
15                     Console.WriteLine("Hello World!");
16                 }
17             }
18         }
19     }
20 }
```

Как вы видите, приложение слегка изменилось.

В 7 строке анализируется длина массива `args`, в котором передаются параметры запуска приложения или, другими словами, аргументы командной строки. Мы сравниваем `args.length` (свойство массива `args`, хранящее длину этого массива) с “1” и если результат сравнения истинен, то это значит, что в нашу программу при запуске был передан 1 аргумент.

В 9 строке мы копируем 0 элемент (индексация массивов начинается с нуля и, значит, первый элемент массива на языке C# будет нулевым) массива `args` в переменную с именем **`name`**.

В 10 строке мы создаем новую строку и подставляем полученное ранее имя в шаблон со словом “Привет {**`name`**}!”.

В 11 строке мы выводим на печать в консоль сконструированную ранее строку.

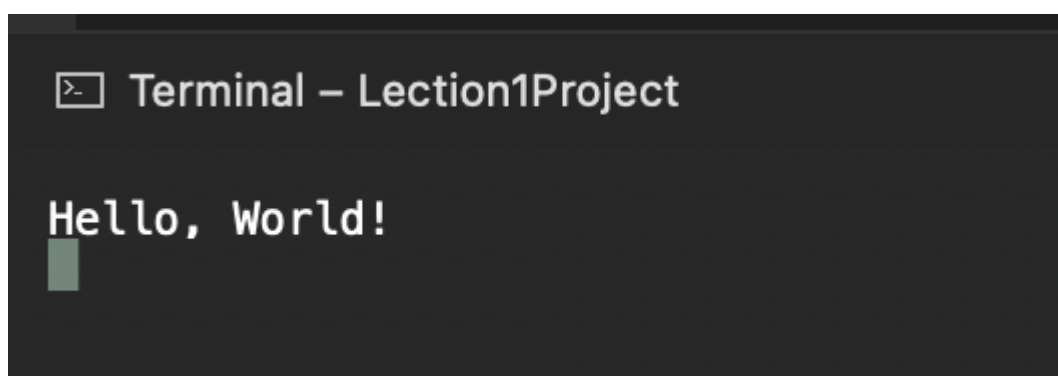
Если же условие в 7 строке ложно (количество аргументов не соответствует 1), то мы просто выводим “Hello, World!” (15 строка).

Итак, мы написали и разобрали работу простого приложения, которое умеет получать параметры (имя) при своем запуске и выводить приветствие с использованием полученных параметров.

Программа использует условный оператор, чтобы обработать сценарий с отсутствием входного параметра.

Запуск и отладка приложения

Запустите приложение через меню [Debug](#)->Start Without Debugging.

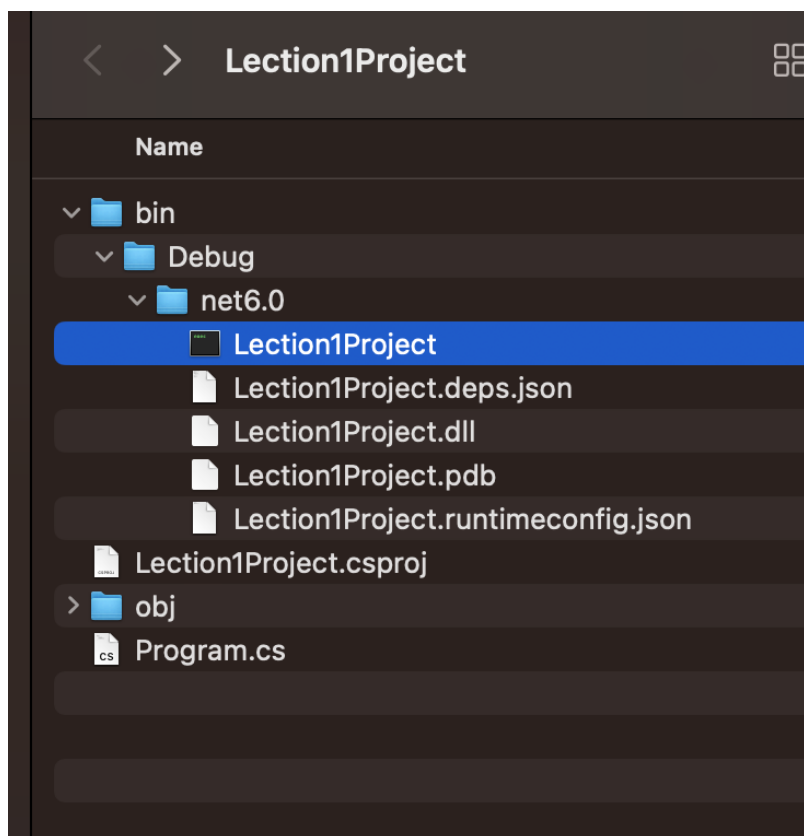


Если все сделано правильно, то в терминале должно отобразиться “Hello, World!”.

Очевидно, что выполнение программы пошло по сценарию “без аргументов”.

Проверить, как работает передача аргументов в наше приложение, можно несколькими способами.

1. Поскольку вы уже запускали приложение, то его исполняемый файл вы сможете найти в подкаталоге bin каталога с проектом.



Откройте терминал в этом каталоге и попробуйте запустить Lecture1Project без параметра и потом с одним параметром.

💡 Параметры (аргументы) командной строки - это мощный инструмент управления, позволяющий определять поведение программы еще до ее старта. Параметры указываются через пробел. Количество параметров, которые вы можете передавать через командную строку, ограничено лишь вашим замыслом. В консольном приложении все параметры, переданные при запуске приложения, будут доступны в виде элементов массива `args`: `args[0]` - первый параметр, `args[1]` - второй параметр и т.д.

Например, если бы утилита для работы с git была написана на C#, то при вызове:

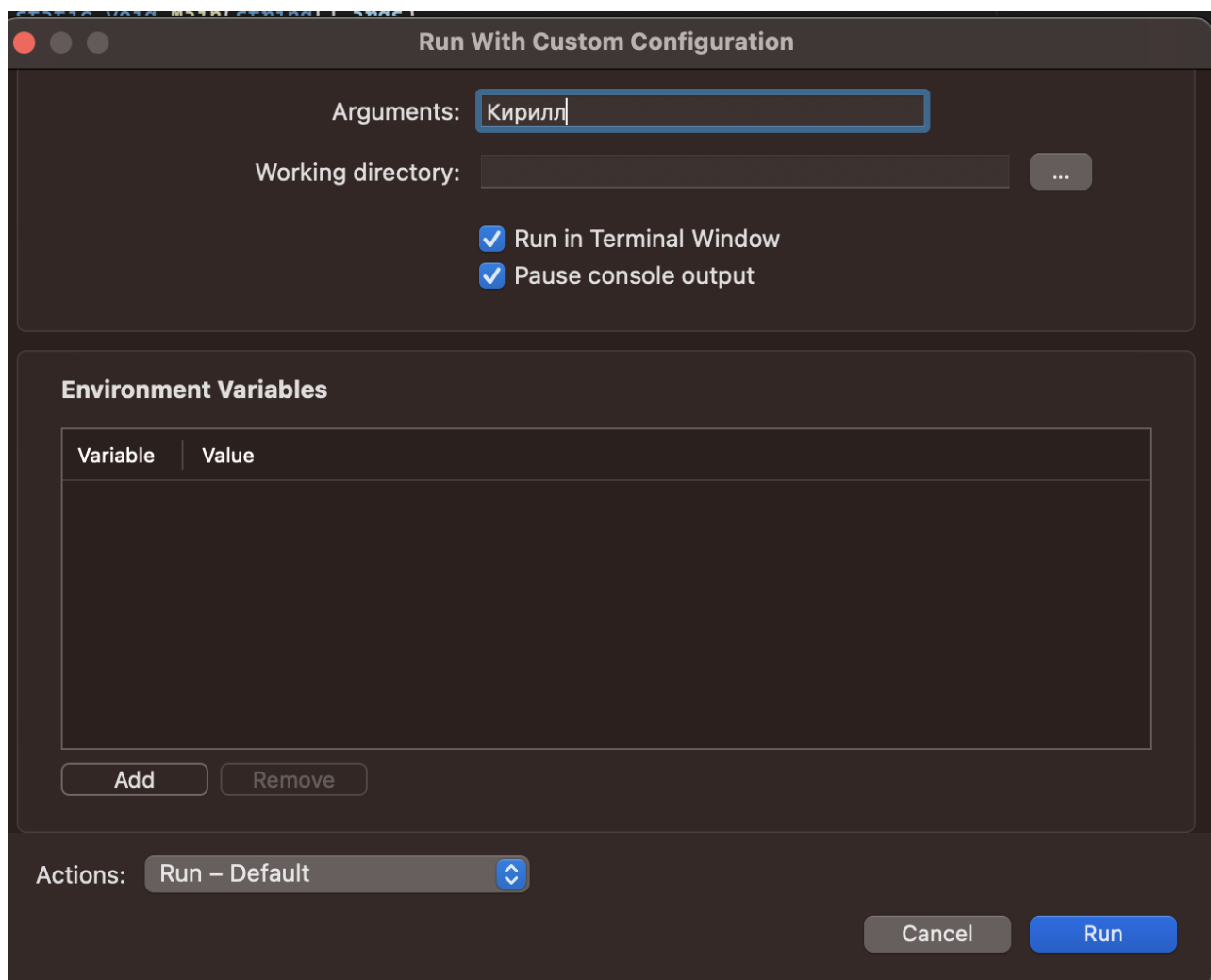
`git clone https://github.com/Jesting/csc.git`

в массиве `args` аргумент `args[0]` был бы равен "clone", а `args[1]` - "https://github.com/Jesting/csc.git".

```
[korlov@Kirills-MacBook-Pro net6.0 % ./Lectio1Project
Hello, World!
[korlov@Kirills-MacBook-Pro net6.0 % ./Lectio1Project Кирилл
Привет Кирилл!
korlov@Kirills-MacBook-Pro net6.0 % █
```

Результат будет выглядеть примерно так, как на этой картинке.

2. Откройте пункт меню [Debug](#)->Run with->Custom configuration.



В появившемся окне введите аргументы и нажмите Run.

Приложение выполнится во встроенном терминале Visual Studio.

```
Terminal – Lektion1Project

Привет Кирилл !
```

Теперь давайте запустим приложение в режиме отладки.

Для этого поставим точку останова напротив первой инструкции в функции Main.

```
4 static void Main(string[] args)
5 {
6     if (args.Length == 1)
7     {
```

И выберем пункт меню [Debug](#)->Start debugging.

Приложение должно начать выполняться, остановившись на поставленной точке останова.



```
1 namespace MyProject;
2 class Program
3 {
4     static void Main(string[] args)
5     {
6         if (args.Length == 1)
7         {
8             string name = args[0];
9             string message = $"Привет {name}!";
10            Console.WriteLine(message);
11        }
12        else
13            Console.WriteLine("Hello, World!");
14    }
```

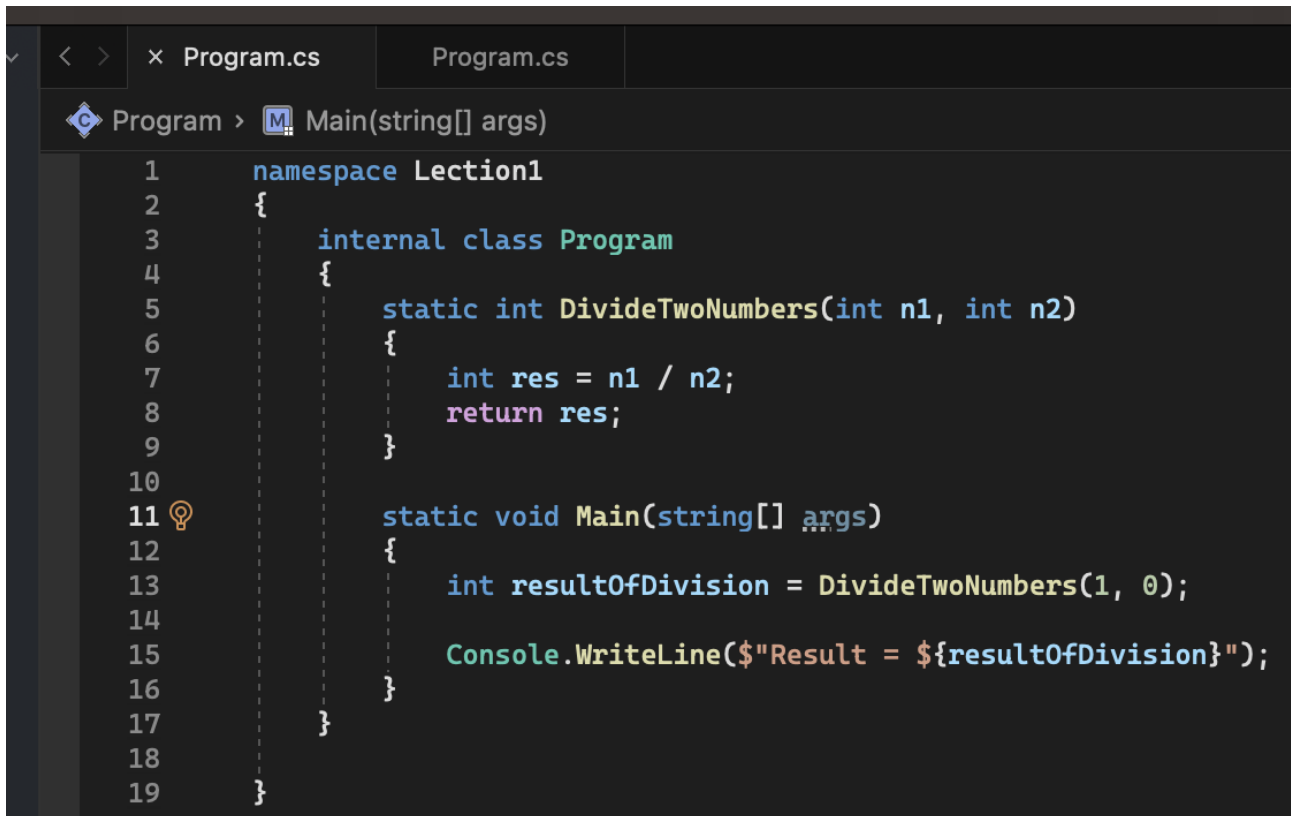
💡 В режиме отладки, когда произошла остановка на точке останова, вы можете посмотреть значение переменных в зоне видимости функции, наведя на них курсор.

Теперь выполним приложение пошагово с помощью пункта меню [Debug](#)->Step Over.

Вы увидите как каждое нажатие приводит к переходу отладчика на следующую выполняемую инструкцию.

Очевидно, что написанное нами приложение работает без ошибок, тогда как процесс отладки необходим в тех случаях, когда приложение работает не так, как мы этого хотим.

Давайте потренируемся на приложении, которое написано с ошибкой.



The screenshot shows a code editor with a file named 'Program.cs'. The code is as follows:

```
1 namespace Lection1
2 {
3     internal class Program
4     {
5         static int DivideTwoNumbers(int n1, int n2)
6         {
7             int res = n1 / n2;
8             return res;
9         }
10
11        static void Main(string[] args)
12        {
13            int resultOfDivision = DivideTwoNumbers(1, 0);
14            Console.WriteLine($"Result = ${resultOfDivision}");
15        }
16    }
17 }
18
19 }
```

A yellow lightbulb icon is placed next to line 11, indicating a warning or error. The code contains a division-by-zero error in the `Main` method where `DivideTwoNumbers(1, 0)` is called.

Задание: попробуйте самостоятельно найти ошибку в этом приложении, проанализировав его код.



Ответ: проблема в делении на ноль. Математика говорит, что такая операция невозможна, так как дает неопределенный результат.

Попробуем запустить программу и посмотрим, что из этого получится.

```
Terminal - Lesson1Program2
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.
   at Lesson1.Program.DivideTwoNumbers(Int32 n1, Int32 n2) in /Users/korlov/csac/lessons/Lesson1/Lesson1Program2/Program.cs:line 7
   at Lesson1.Program.Main(String[] args) in /Users/korlov/csac/lessons/Lesson1/Lesson1Program2/Program.cs:line 13
/bin/bash: line 1: 98708 Abort trap: 6           "/usr/local/share/dotnet/dotnet" "/Users/korlov/csac/lessons/Lesson1/Lesson1Program2/bin/Debug/net7.0/Lesson1Program2.dll"
```

Результатом выполнения программы стала ошибка. Давайте разберем, что она значит.

Unhandled exception означает, что возникшая в программе ошибка не была обработана самой программой и среде выполнения не оставалось ничего кроме как аварийно завершить программу. В ходе нашего курса мы, конечно же, научимся обрабатывать такие ошибки.

Далее уточняется класс исключения: **System.DivideByZeroException**, по названию которого легко понять, что это ошибка деления на ноль.

Далее идет текстовое описание ошибки: **Attempted to divide by zero**, подтверждающее нашу версию.

На последующих строках располагается **стек вызова** (последовательность вызовов функций, приведших к ошибке).

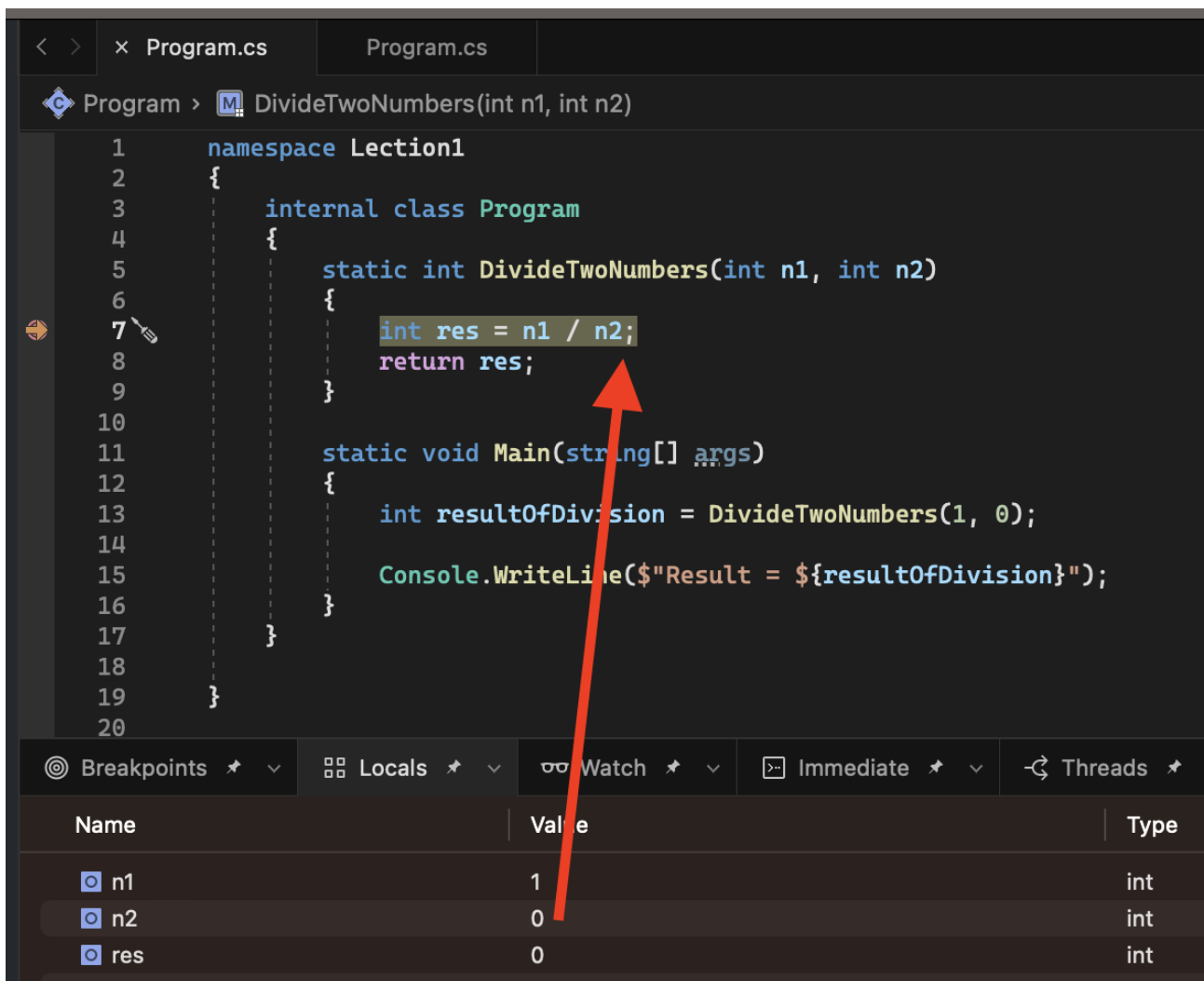
line 7 - непосредственно то место в функции, где произошла ошибка.

line 13 - место, откуда был произведен вызов "ошибочной" функции.

Стек вызовов заканчивается строкой, говорящий о том, что предыдущий вызов является запуском самого приложения. Поэтому строка не указана.

Из сообщения об ошибке становится понятно, на какой строке кода произошла ошибка. Но пока еще не понятно, что именно ее вызвало.

Давайте поставим точку останова на строку 7 и запустим отладчик:



```
1 namespace Lektion1
2 {
3     internal class Program
4     {
5         static int DivideTwoNumbers(int n1, int n2)
6         {
7             int res = n1 / n2;
8             return res;
9         }
10
11         static void Main(string[] args)
12         {
13             int resultOfDivision = DivideTwoNumbers(1, 0);
14             Console.WriteLine($"Result = ${resultOfDivision}");
15         }
16     }
17 }
18
19
20
```

Name	Value	Type
n1	1	int
n2	0	int
res	0	int

После остановки программы на точке, среда покажет вам значения локальных переменных (если окна Locals у вас нет, вы можете открыть его через меню Debug->Windows->Locals).

Посмотрев на значения переменных, становится понятно, что проблема в переменной n2, играющей роль делителя: ее значение равно нулю и именно это приводит к ошибке.

Разумеется, в нашем примере мы знали о причинах ошибке заранее, так как весь код удобно располагался на одном экране и можно было визуальнo отследить всю последовательность вызовов, найдя ошибку без отладки.

В больших проектах или же сложных алгоритмах, где количество шагов программы равняется десяткам или даже сотням, отладка становится незаменимым инструментом поиска ошибок.

Стиль написания кода

При работе в команде над проектом, важно чтобы коллеги не тратили лишнее время, пытаясь разобраться в коде, который вы написали.

Этого можно добиться комплексом мер, одной из которых является унифицирование стиля, в котором пишется код. У Microsoft есть официальное руководство о том, как правильно писать код.

Ниже на скриншоте наш код с делением на ноль приведен таким образом, что он стал более компактным, не потеряв способности выполняться - его по-прежнему можно скомпилировать и запустить.

Теперь он занимает всего 7 строк.

```
namespace MyProject;
class Program{
static int MY_function(int n1, int n2){//МОЯ ФУНКЦИЯ
int res = n1 /n2;return res;}
static void Main(string[] args){
int my_____VARIABLE = MY_function(1, 1);//ВЫЗОВ МОЕЙ ФУНКЦИИ И ЗАПИСЬ ЕЕ ЗНАЧЕНИЯ В ПЕРЕМЕННУЮ
Console.WriteLine($"Result = {my_____VARIABLE}");}}
```

Стал ли код понятнее? Очевидно, что ответ нет.

Ваш коллега по работе будет сильно расстроен, если ему придется искать ошибку в таком коде и, скорее всего, он заставит вас переписать его с нуля.

Давайте разберем, почему этот код сложно понять:

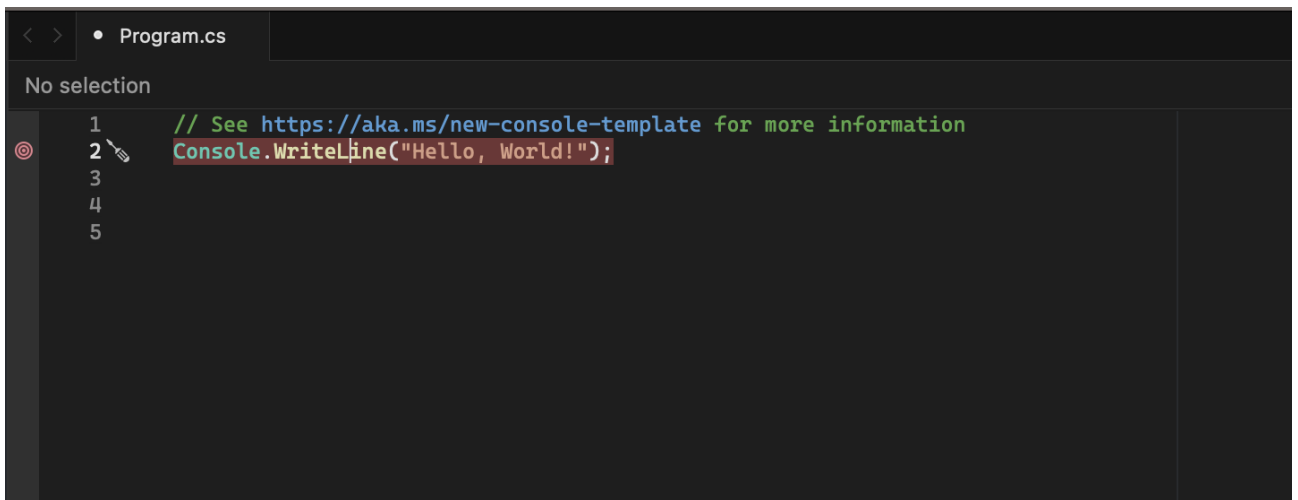
- **Отступы.** Вложенные блоки кода должны отделяться горизонтальной табуляцией, чтобы быть визуально отличимыми. Каждый новый уровень вложенности добавляет одну табуляцию ко всему коду на этом уровне.

```
if (УСЛОВИЕ)
{
    if (ВЛОЖЕННОЕ_УСЛОВИЕ)
    {
        //ВАШ КОД
    }
}
```

- **Одна строка - одна команда.** Не размещайте несколько инструкций на одной строке.
- **Имена функций и переменных должны отражать их назначение.** Если переименовать MY_function в divideTwoNumbers, то сразу становится понятно,

что делает эта функция. Пишите названия функций так, чтобы по названию было понятно ее предназначение.

- **Комментарии.** Хорошо написанный код не нуждается в комментариях. Не стоит комментировать очевидные вещи. Чаще всего правильное именование переменных и функций дает полное представление о коде. Иногда все же комментарии необходимы. Например, код, который создается по шаблону консольного приложения в последней версии Visual Studio, содержит комментарий, объясняющий, почему шаблон проекта сократился до одной строки.



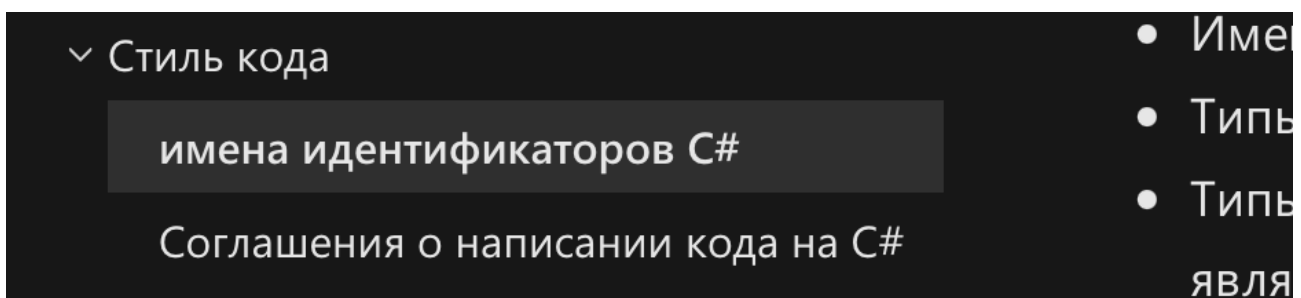
```
< > • Program.cs
No selection
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
4
5
```

Старайтесь писать функции таким образом, чтобы ее код помещался в один экран по длине и ширине. Очевидно, что у разных разработчиков параметры длины и ширины экрана могут отличаться, поэтому ориентируйтесь исключительно на свои.

У Microsoft есть подробный гайд по тому, как должен выглядеть соответствующий стандарту код.

В этом гайде вы найдете подробное руководство о правильном именовании всех элементов языка в статьях: “имена идентификаторов C#” и “Соглашения о написании кода”. Изучите руководство заранее. Это избавит вас от проблем с коллегами и упростит вашу собственную работу.

Гайд: [Имена идентификаторов C# | Microsoft Learn](#)



- ▼ Стиль кода
 - имена идентификаторов C#
 - Соглашения о написании кода на C#
- Име...
- Тип...
- Тип...
- явля...

Заключение

На этом уроке мы разобрали устройство платформы .Net. Рассмотрели историю разработки платформы, ее версии и их отличия. Подробно остановились на составных частях .Net, таких как FCL и CLR.

В ходе урока мы изучили инструменты Visual Studio, используемые для создания и отладки приложений. А также научились устанавливать Visual Studio и погрузились в меню ее инструментов.

Также на уроке мы выяснили, какие основные виды приложений существуют и для чего предназначено каждое из них. Нам стало понятно, как создать приложение на основе шаблона проекта.

Еще мы изучили структуру .Net-проекта и назначение отдельных его файлов. Разобрались что такое “проект” и что такое “решение”.

Кроме того, мы написали наше первое приложение при помощи Visual Studio. Получили представление о структуре программы и научились передавать в нее параметры через командную строку.

Во время работы с кодом мы учились использовать отладчик. Применили его для поиска ошибки в написанном коде.

Немаловажно, что мы узнали о стиле написания кода. Определили, почему стиль написания кода важен как для нас самих, так и для коллег по команде. Во время этой работы мы выявили типовые ошибки при оформлении кода.

Итак, нам удалось достичь цели урока: мы поняли, как устроена платформа .Net, как работают ее основные компоненты и какие инструменты мы будем применять в работе над проектами в дальнейших уроках и на практике.

Термины, используемые в лекции

CLR - Common Language Runtime, среда выполнения .Net приложений.

Управляемый код / managed code - код, выполняемый под управлением CLR.

Неуправляемый код/ unmanaged code - код, выполняемый за пределами CLR.

IL - intermediate language, промежуточный код, в который компилируется приложение.

JIT - just in time compiler - компонент CLR, программа, преобразующая коды IL в машинные коды в ходе выполнения приложения.

Debug – процесс отладки приложения / версия приложения, используемая разработчиком для поиска ошибок или оптимизации работы используемых алгоритмов.

Сборка мусора/garbage collection - автоматический процесс освобождения памяти.

Solution /решение - инструмент группировки, объединяющий в себе один или несколько проектов.

Unit-test/юнит-тест - функция проверяющая работоспособность небольшой части кода.

Open source code / открытый исходный код /открытое ПО - разновидность программного обеспечения с исходными кодами, доступными публично.

FCL (Framework classes library) - библиотека классов .Net.

Домашнее задание

- 1) Установите редактор кода Visual Studio (<https://visualstudio.microsoft.com/>).
- 2) Изучите рекомендации по стилю написания кода на официальном сайте Microsoft: [Имена идентификаторов C# | Microsoft Learn](#).
- 3) Попробуйте выполнить задание на основе второго приложения (с ошибкой деления на ноль). Перепишите функцию деления двух чисел таким образом, чтобы она:
 - а) предупреждала пользователя об ошибке,
 - б) не выполняла деление, в случае если делитель равен нулю.



Домашнее задание по лекциям не проверяется, но может быть адресно рассмотрено в ходе семинара.

Используемая литература

<https://learn.microsoft.com/> - здесь вы найдете информацию по установке Visual Studio, типам проектов и дополнительно сможете изучить все возможности среды разработки.

Вы можете просмотреть темы “Создание консольного приложения” и “Отладка приложения” для закрепления материала.

Этот текст может быть частично переведен средствами машинного перевода.

Фильтровать по названию

Learn / .NET / Основные принципы .NET / C#

Учебник. Создание консольного приложения .NET в Visual Studio

Статья • 09.12.2022 • Чтение занимает 6 мин • Участники: 17 [Обратная связь](#)

Выберите версию .NET

[.NET 7](#) [.NET 6](#) [.NET 5](#) [.NET Core 3.1](#)

В этом руководстве показано, как создать и запустить консольное приложение .NET с помощью Visual Studio 2022.

Предварительные требования

- Visual Studio 2022 версии 17.4 или более поздней с установленной рабочей нагрузкой **разработка классических приложений .NET**. Пакет SDK для .NET 7 устанавливается автоматически при выборе этой рабочей нагрузки.

См. раздел [Установка пакета SDK для .NET с помощью Visual Studio](#).

Создание приложения

Документация по .NET

- > Начало работы
- > Обзор
 - Введение в .NET
 - Реализации .NET
 - Библиотеки классов .NET
 - Обзор .NET Standard
 - Выпуски, исправления и поддержка
 - Стандарты ECMA
 - Глоссарий по .NET
- > Учебники
 - Изменение шаблонов .NET 6
 - Использование Visual Studio**
 - Создание консольного приложения**
 - Отладка приложения
 - Публикация приложения
 - Создание библиотеки
 - Модульное тестирование библиотеки
 - Установка и использование пакета
 - Создание и публикация пакета

Ссылка на гитхаб

Программы из этой лекции: <https://github.com/Jesting/csc/tree/dev/Lecture1>