*Увлекательны023*

*й C# - как язык*

*Стич Назар ИВТ-22 Объектно Ориентированное Программирование*

[*https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/*](https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/)

[*https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/program-structure/*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/program-structure/)

[*https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/tutorial-console?view=vs-2022*](https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/tutorial-console?view=vs-2022)

[*https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/)

[*https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/classes*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/classes)

[*https://learn.microsoft.com/ru-ru/dotnet/core/testing/*](https://learn.microsoft.com/ru-ru/dotnet/core/testing/)

[*https://uproger.com/dorozhnaya-karta-razrabotchikov-net-v-2024-godu-dlya-nachinayushhih/*](https://uproger.com/dorozhnaya-karta-razrabotchikov-net-v-2024-godu-dlya-nachinayushhih/)

*Краткий обзор языка C#*

C# (произносится как "си шарп") — современный объектно-ориентированный и типобезопасный язык программирования. C# позволяет разработчикам создавать разные типы безопасных и надежных приложений, выполняющихся в .NET. C# относится к широко известному семейству языков C, и покажется хорошо знакомым любому, кто работал с C, C++, Java или JavaScript. Здесь представлен обзор основных компонентов языка C# 8 и более ранних версий. Если вы хотите изучить язык с помощью интерактивных примеров, рекомендуем поработать с [вводными руководствами по C#](https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/tutorials/).

C# — объектно-ориентированный, *ориентированный на компоненты* язык программирования. C# предоставляет языковые конструкции для непосредственной поддержки такой концепции работы. Благодаря этому C# подходит для создания и применения программных компонентов. С момента создания язык C# обогатился функциями для поддержки новых рабочих нагрузок и современными рекомендациями по разработке ПО. В основном C# — *объектно-ориентированный* язык. Вы определяете типы и их поведение.

Вот лишь несколько функций языка C#, которые позволяют создавать надежные и устойчивые приложения. [*Сборка мусора*](https://learn.microsoft.com/ru-ru/dotnet/standard/garbage-collection/) автоматически освобождает память, занятую недостижимыми неиспользуемыми объектами. [*Типы, допускающие значение null,*](https://learn.microsoft.com/ru-ru/dotnet/csharp/nullable-references) обеспечивают защиту от переменных, которые не ссылаются на выделенные объекты. [*Обработка исключений*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/exceptions/) предоставляет структурированный и расширяемый подход к обнаружению ошибок и восстановлению после них. [*Лямбда-выражения*](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/lambda-expressions) поддерживают приемы функционального программирования. [*Синтаксис LINQ*](https://learn.microsoft.com/ru-ru/dotnet/csharp/linq/) создает общий шаблон для работы с данными из любого источника. Поддержка языков для [*асинхронных операций*](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/) предоставляет синтаксис для создания распределенных систем. В C# имеется [*Единая система типов*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/). Все типы C#, включая типы-примитивы, такие как int и double, наследуют от одного корневого типа object. Все типы используют общий набор операций, а значения любого типа можно хранить, передавать и обрабатывать схожим образом. Более того, C# поддерживает как определяемые пользователями [ссылочные типы](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types), так и [типы значений](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/value-types). C# позволяет динамически выделять объекты и хранить упрощенные структуры в стеке. C# поддерживает универсальные методы и типы, обеспечивающие повышенную безопасность типов и производительность. C# предоставляет итераторы, которые позволяют разработчикам классов коллекций определять пользовательские варианты поведения для клиентского кода.

C# подчеркивает *Управление версиями* , чтобы обеспечить совместимость программ и библиотек с течением времени. Вопросы управления версиями существенно повлияли на такие аспекты разработки C#, как раздельные модификаторы virtual и override, правила разрешения перегрузки методов и поддержка явного объявления членов интерфейса.

Архитектура .NET

Программы C# выполняются в .NET, виртуальной системе выполнения, вызывающей общеязыковую среду выполнения (CLR) и набор библиотек классов. Среда CLR — это реализация общеязыковой инфраструктуры языка (CLI), являющейся международным стандартом, от корпорации Майкрософт. CLI является основой для создания сред выполнения и разработки, в которых языки и библиотеки прозрачно работают друг с другом.

Исходный код, написанный на языке C# компилируется в [промежуточный язык (IL)](https://learn.microsoft.com/ru-ru/dotnet/standard/managed-code), который соответствует спецификациям CLI. Код на языке IL и ресурсы, в том числе растровые изображения и строки, сохраняются в сборке, обычно с расширением *.dll*. Сборка содержит манифест с информацией о типах, версии, языке и региональных параметрах для этой сборки.

При выполнении программы C# сборка загружается в среду CLR. Среда CLR выполняет JIT-компиляцию из кода на языке IL в инструкции машинного языка. Среда CLR также выполняет другие операции, например, автоматическую сборку мусора, обработку исключений и управление ресурсами. Код, выполняемый средой CLR, иногда называют "управляемым кодом". "Неуправляемый код" компилируется на машинный язык, предназначенный для конкретной платформы.

Обеспечение взаимодействия между языками является ключевой особенностью .NET. Код IL, созданный компилятором C#, соответствует спецификации общих типов (CTS). Код IL, созданный из кода на C#, может взаимодействовать с кодом, созданным из версий .NET для языков F#, Visual Basic, C++. Существует более 20 других языков, совместимых с CTS. Одна сборка может содержать несколько модулей, написанных на разных языках .NET, и все типы могут ссылаться друг на друга, как если бы они были написаны на одном языке.

В дополнение к службам времени выполнения .NET также включает расширенные библиотеки. Эти библиотеки поддерживают множество различных рабочих нагрузок. Они упорядочены по пространствам имен, которые предоставляют разные полезные возможности: от операций файлового ввода и вывода до управления строками и синтаксического анализа XML, от платформ веб-приложений до элементов управления Windows Forms. Обычно приложение C# активно используют библиотеку классов .NET для решения типовых задач.

Дополнительные сведения о .NET, см. в статье [Обзор .NET](https://learn.microsoft.com/ru-ru/dotnet/core/introduction).

Здравствуй, мир

Для первого знакомства с языком программирования традиционно используется программа "Hello, World". Вот ее пример на C#:

C#Копировать

Выполнить

using System;

class Hello

{

static void Main()

{

Console.WriteLine("Hello, World");

}

}

Программа "Hello, World" начинается с директивы using, которая ссылается на пространство имен System. Пространства имен позволяют иерархически упорядочивать программы и библиотеки C#. Пространства имен содержат типы и другие пространства имен. Например, пространство имен System содержит несколько типов (в том числе используемый в нашей программе класс Console) и несколько других пространств имен, таких как IO и Collections. Директива using, которая ссылается на пространство имен, позволяет использовать типы из этого пространства имен без указания полного имени. Благодаря директиве using в коде программы можно использовать сокращенное имя Console.WriteLine вместо полного варианта System.Console.WriteLine.

Класс Hello, объявленный в программе "Hello, World", имеет только один член — это метод с именем Main. Метод Main объявлен с модификатором static. Методы экземпляра могут ссылаться на конкретный экземпляр объекта, используя ключевое слово this, а статические методы работают без ссылки на конкретный объект. По стандартному соглашению точкой входа программы C# является статический метод с именем Main.

Выходные данные программы создаются в методе WriteLine класса Console из пространства имен System. Этот класс предоставляется библиотеками стандартных классов, ссылки на которые компилятор по умолчанию добавляет автоматически.

Типы и переменные

*Тип* определяет структуру и поведение любых данных в C#. Объявление типа может включать его члены, базовый тип, интерфейсы, которые он реализует, и операции, разрешенные для этого типа. *Переменная* — это метка, которая ссылается на экземпляр определенного типа.

В C# существуют две разновидности типов: *ссылочные типы* и *типы значений*. Переменные типа значений содержат непосредственно данные, а в переменных ссылочных типов хранятся ссылки на нужные данные, которые именуются объектами. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому может случиться так, что операции над одной переменной затронут объект, на который ссылается другая переменная. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров ref и out).

*Идентификатор* — это имя переменной. Идентификатор — это последовательность символов Юникода без пробелов. Идентификатор может быть зарезервированным словом C#, если он имеет префикс @. При взаимодействии с другими языками в качестве идентификатора может быть полезно использовать зарезервированное слово.

Типы значений в C# делятся на *простые типы*, *типы перечислений*, *типы структур*, *типы, допускающие значение NULL*, и *типы значений кортежей*. Ссылочные типы в C# подразделяются на *типы классов*, *типы интерфейсов*, *типы массивов* и *типы делегатов*.

Далее представлены общие сведения о системе типов в C#.

[Типы значений](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/value-types)

[Простые типы](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/value-types#built-in-value-types)

[Целое со знаком](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): , short , int , long

[Целое без знака](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): , ushort , uint , ulong

[Символы Юникода](https://learn.microsoft.com/ru-ru/dotnet/standard/base-types/character-encoding-introduction): , представляющие блок кода UTF-16

[Двоичная с плавающей запятой IEEE](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types): , double

[Десятичная точность с плавающей запятой с высокой точностью](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types):

Логический: bool, используется для представления логических значений, которые могут иметь значение true или false.

[Типы перечисления](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/enum)

Пользовательские типы в формате enum E {...}. Тип enum является отдельным типом со списком именованных констант. Каждый тип enum имеет базовый тип, в роли которого выступает один из восьми целочисленных типов. Набор значений типа enum аналогичен набору значений его базового типа.

[Типы структур](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct)

Пользовательские типы в формате struct S {...}

[Типы значений, допускающие значение NULL](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nullable-value-types)

Расширения других типов значений, допускающие значение null

[Типы значений кортежей](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/value-tuples)

Пользовательские типы в формате (T1, T2, ...)

[Ссылочные типы](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/reference-types)

[Типы классов](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class)

Исходный базовым классом для всех типов: object

[Строки в Юникоде](https://learn.microsoft.com/ru-ru/dotnet/standard/base-types/character-encoding-introduction): , представляющие последовательность единиц кода UTF-16

Пользовательские типы в формате class C {...}

[Типы интерфейсов](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/interface)

Пользовательские типы в формате interface I {...}

[Типы массивов](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/arrays/)

Одномерные, многомерные массивы и массивы массивов. Например, int[], int[,] и int[][].

[Типы делегатов](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types#the-delegate-type)

Пользовательские типы в формате delegate int D(...)

Программы C# используют *объявления типов* для создания новых типов. В объявлении типа указываются имя и члены нового типа. Шесть категорий типов в C# определяются пользователем: типы классов, типы структур, типы интерфейсов, типы перечисления, типы делегатов и типы значений кортежей. Можно также объявлять типы record, либо record struct, либо record class. Типы записей имеют члены, синтезированные компилятором. Записи используются в основном для хранения значений с минимальным связанным поведением.

Тип class определяет структуру данных, которая содержит данные-члены (поля) и функции-члены (методы, свойства и т. д.). Классы поддерживают механизмы одиночного наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие и уточняющие определения базовых классов.

Тип struct похож на тип класса тем, что он представляет структуру с данными-членами и функциями-членами. Но, в отличие от классов, структуры являются типами значений и обычно не требуют выделения памяти из кучи. Типы структуры не поддерживают определяемое пользователем наследование, и все типы структуры неявно наследуют от типа object.

Тип interface определяет контракт в виде именованного набора открытых элементов. Объект типа class или struct, реализующий interface, должен предоставить реализации для всех элементов интерфейса. Тип interface может наследовать от нескольких базовых интерфейсов, а class или struct могут реализовывать несколько интерфейсов.

Тип delegate (делегат) представляющий ссылки на методы с конкретным списком параметров и типом возвращаемого значения. Делегаты позволяют использовать методы как сущности, сохраняя их в переменные и передавая в качестве параметров. Делегаты аналогичны типам функций, которые используются в функциональных языках. Их принцип работы близок к указателям функций из некоторых языков. В отличие от указателей функций, делегаты являются объектно-ориентированными и типобезопасными.

Типы class, struct, interface и delegate поддерживают универсальные шаблоны, которые позволяют передавать им другие типы в качестве параметров.

C# поддерживает одномерные и многомерные массивы любого типа. В отличие от перечисленных выше типов, типы массивов не требуется объявлять перед использованием. Типы массивов можно сформировать, просто введя квадратные скобки после имени типа. Например, int[] является одномерным массивом значений типа int, а int[,] — двумерным массивом значений типа int, тогда как int[][] представляет собой одномерный массив одномерных массивов (или массив массивов) значений типа int.

Типы, допускающие значение NULL, не требуют отдельного определения. Для каждого обычного типа T, который не допускает значение NULL, существует идентичный тип T?, который отличается только тем, что может содержать дополнительное значение null. Например, int? является типом, который может содержать любое 32-разрядное целое число или значение null, а string? — любое значение string или null.

Система типов в C# унифицирована таким образом, что значение любого типа можно рассматривать как object (объект). Каждый тип в C# является прямо или косвенно производным от типа класса object, и этот тип object является исходным базовым классом для всех типов. Чтобы значения ссылочного типа обрабатывались как объекты, им просто присваивается тип object. Чтобы значения типов значений обрабатывались как объекты, выполняются операции *упаковки-преобразования* и *распаковки-преобразования*. В следующем примере значение int преобразуется в object, а затем обратно в int.

C#Копировать

int i = 123;

object o = i; // Boxing

int j = (int)o; // Unboxing

Если значение типа назначается ссылке object, для хранения значения выделяется упаковка. Эта упаковка является экземпляром ссылочного типа, и в нее копируется значение. И наоборот, если ссылка типа object используется для типа значения, для соответствующего object выполняется проверка, является ли он упаковкой правильного типа. Если эта проверка завершается успешно, копируется значение этой упаковки.

Унифицированная система типов C# фактически означает, что типы значений обрабатываются как object ссылки "по запросу". Из-за унификации библиотеки общего назначения, использующие тип object , могут использоваться со всеми типами, производными от object , включая ссылочные типы и типы значений.

В C# существует несколько типов *переменных*, в том числе поля, элементы массива, локальные переменные и параметры. Переменные представляют собой места хранения, и каждая переменная имеет тип, который определяет допустимые значения для хранения в этой переменной. Примеры представлены ниже.

Тип значения, не допускающий значения Null

Значение такого типа

Тип значения, допускающий значение Null

Значение null или значение такого типа

object

Ссылка null, ссылка на объект любого ссылочного типа или ссылка на упакованное значение любого типа значения

Тип класса

Ссылка null, ссылка на экземпляр такого типа класса или ссылка на экземпляр любого класса, производного от такого типа класса

Тип интерфейса

Ссылка null, ссылка на экземпляр типа класса, который реализует такой тип интерфейса, или ссылка на упакованное значение типа значения, которое реализует такой тип интерфейса

Тип массива

Ссылка null, ссылка на экземпляр такого типа массива или ссылка на экземпляр любого совместимого типа массива

Тип делегата

Ссылка null или ссылка на экземпляр совместимого типа делегата

Структура программы

Основными понятиями Организации в C# являются [*программы*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/program-structure/), [*пространства имен*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/namespaces), [*типы*](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/), [*члены*](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/members)и [*сборки*](https://learn.microsoft.com/ru-ru/dotnet/standard/assembly/). В программе объявляются типы, которые содержат члены. Эти типы можно организовать в пространства имен. Примерами типов являются классы, структуры и интерфейсы. К членам относятся поля, методы, свойства и события. При компиляции программы на C# упаковываются в сборки. Сборки обычно имеют расширение .exe файла или .dll , в зависимости от того, реализуют ли они .exe или *библиотеки*соответственно.

В качестве небольшого примера рассмотрим сборку, содержащую следующий код:

C#Копировать

namespace Acme.Collections;

public class Stack<T>

{

Entry \_top;

public void Push(T data)

{

\_top = new Entry(\_top, data);

}

public T Pop()

{

if (\_top == null)

{

throw new InvalidOperationException();

}

T result = \_top.Data;

\_top = \_top.Next;

return result;

}

class Entry

{

public Entry Next { get; set; }

public T Data { get; set; }

public Entry(Entry next, T data)

{

Next = next;

Data = data;

}

}

}

Полное имя этого класса: Acme.Collections.Stack. Этот класс содержит несколько членов: поле с именем \_top, два метода с именами Push и Pop, а также вложенный класс с именем Entry. Класс Entry, в свою очередь, содержит три члена: свойство с именем Next, свойство с именем Data и конструктор. StackЯвляется Stack классом. Он имеет параметр одного типа T, который замещается конкретным типом при использовании.

*Стек* — это коллекция типа FILO (прибыл первым — обслужен последним). Новые элементы добавляются в верх стека. Удаляемый элемент исключается из верхней части стека. В предыдущем примере объявляется тип Stack, который определяет хранилище и поведение для стека. Можно объявить переменную, которая ссылается на экземпляр типа Stack для использования этой возможности.

Сборки содержат исполняемый код в виде инструкций промежуточного языка (IL) и символьную информацию в виде метаданных. Перед выполнением JIT-компилятор среды CLR .NET преобразует код IL в сборке в код, зависящий от процессора.

Сборка полностью описывает сама себя и содержит весь код и метаданные, поэтому в C# не используются директивы #include и файлы заголовков. Чтобы использовать в программе C# открытые типы и члены, содержащиеся в определенной сборке, вам достаточно указать ссылку на эту сборку при компиляции программы. Например, эта программа использует класс Acme.Collections.Stack из сборки acme.dll:

C#Копировать

class Example

{

public static void Main()

{

var s = new Acme.Collections.Stack<int>();

s.Push(1); // stack contains 1

s.Push(10); // stack contains 1, 10

s.Push(100); // stack contains 1, 10, 100

Console.WriteLine(s.Pop()); // stack contains 1, 10

Console.WriteLine(s.Pop()); // stack contains 1

Console.WriteLine(s.Pop()); // stack is empty

}

}

Для компиляции программы вам потребуется создать *ссылку* на сборку, содержащую класс стека, определенный в примере выше.

Программы C# можно хранить в нескольких исходных файлах. При компиляции программы C# все исходные файлы обрабатываются вместе, при этом они могут свободно ссылаться друг на друга. По сути, это аналогично тому, как если бы все исходные файлы были объединены в один большой файл перед обработкой. В C# никогда не используются опережающие объявления, так как порядок объявления, за редким исключением, не играет никакой роли. В C# нет требований объявлять только один открытый тип в одном исходном файле, а также имя исходного файла не обязано совпадать с типом, объявляемом в этом файле.

Такие организационные блоки описываются в других статьях этого обзора.

*Общая структура программы C#*

Программа на языке C# состоит из одного или нескольких файлов. Каждый файл может содержать или не содержать пространства имен. Пространство имен может содержать типы, такие как классы, структуры, интерфейсы, перечисления и делегаты или другие пространства имен. Ниже приведен пример структуры программы на C#, содержащей все эти элементы.

C#Копировать

// A skeleton of a C# program

using System;

// Your program starts here:

Console.WriteLine("Hello world!");

namespace YourNamespace

{

class YourClass

{

}

struct YourStruct

{

}

interface IYourInterface

{

}

delegate int YourDelegate();

enum YourEnum

{

}

namespace YourNestedNamespace

{

struct YourStruct

{

}

}

}

В предыдущем примере для точки входа программы используются инструкции верхнего уровня. Эта функция была добавлена в C# 9. До версии C# 9 точка входа была статическим методом с именем Main, как показано в следующем примере:

C#Копировать

// A skeleton of a C# program

using System;

namespace YourNamespace

{

class YourClass

{

}

struct YourStruct

{

}

interface IYourInterface

{

}

delegate int YourDelegate();

enum YourEnum

{

}

namespace YourNestedNamespace

{

struct YourStruct

{

}

}

class Program

{

static void Main(string[] args)

{

//Your program starts here...

Console.WriteLine("Hello world!");

}

}

}

*Учебник. Создание простого консольного приложения C# в Visual Studio (часть 1 из 2)*

Показать еще 4

Изучая этот учебник, вы создадите и запустите консольное приложение C# с помощью Visual Studio, а также ознакомитесь с некоторыми возможностями интегрированной среды разработки (IDE) Visual Studio. Этот учебник представляет собой первую часть серии, состоящей из двух частей.

В этом руководстве выполняются следующие задачи:

Создание проекта Visual Studio

Создание консольного приложения C#

Запустите отладку приложения.

Закрытие приложения

Проверка готового кода

[В рамках части 2](https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/tutorial-console-part-2?view=vs-2022) вы расширите это приложение, добавив дополнительные проекты, изучите методы отладки и будете использовать сторонние пакеты.

Необходимые компоненты

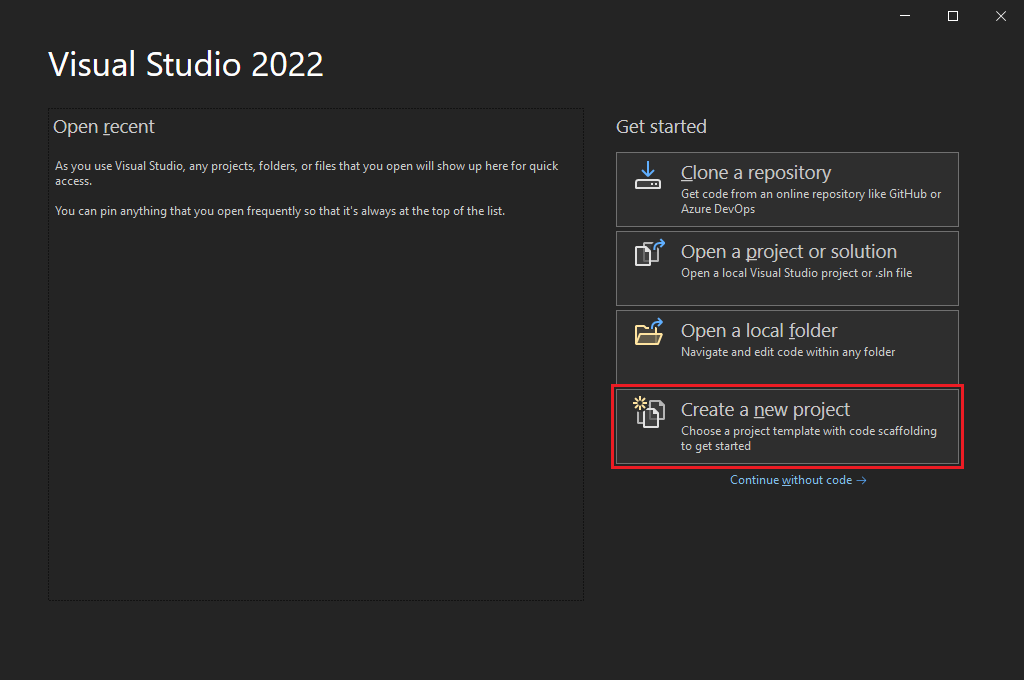
Необходимо установить Visual Studio.

Установите Visual Studio бесплатно со страницы [скачиваемых материалов Visual Studio](https://visualstudio.microsoft.com/downloads/?cid=learn-onpage-download-cta), если еще не сделали этого.

Создание проекта

Сначала создайте проект приложения C#. Для этого типа проекта уже имеются все нужные файлы шаблонов.

Откройте Visual Studio и выберите "Создать проект " в окне "Пуск".

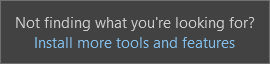


В окне Создать проект выберите Все языки, а затем в раскрывающемся списке выберите C#. Выберите Windows в списке Все платформы. Затем в списке Все типы проектов выберите Консоль.

Применив фильтры по языку, платформе и типу проекта, выберите шаблон Консольное приложение и нажмите Далее.

 Примечание

Если шаблон Консольное приложение не отображается, щелкните Установка других средств и компонентов.



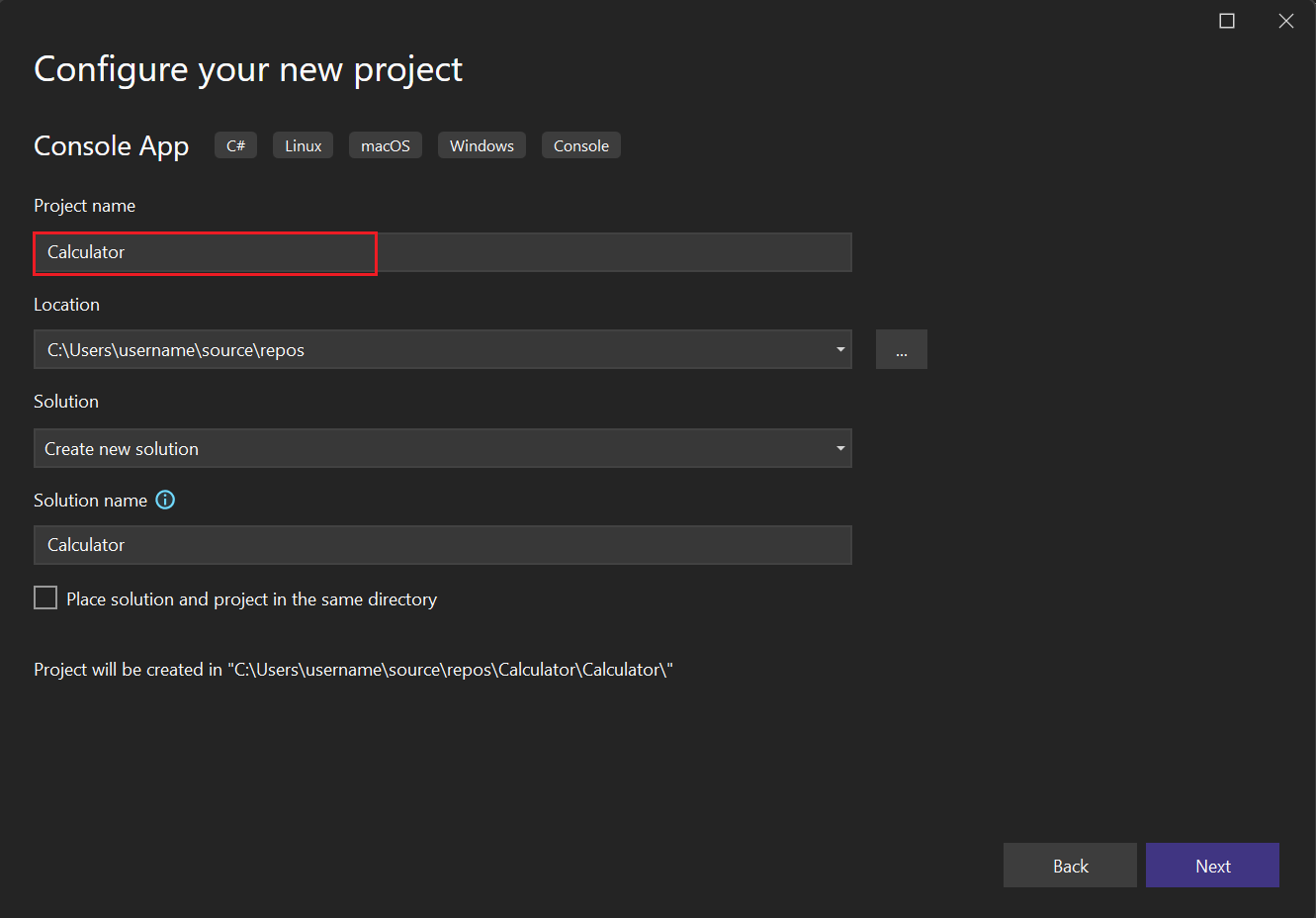
В установщике Visual Studio выберите рабочую нагрузку разработки классических приложений .NET.



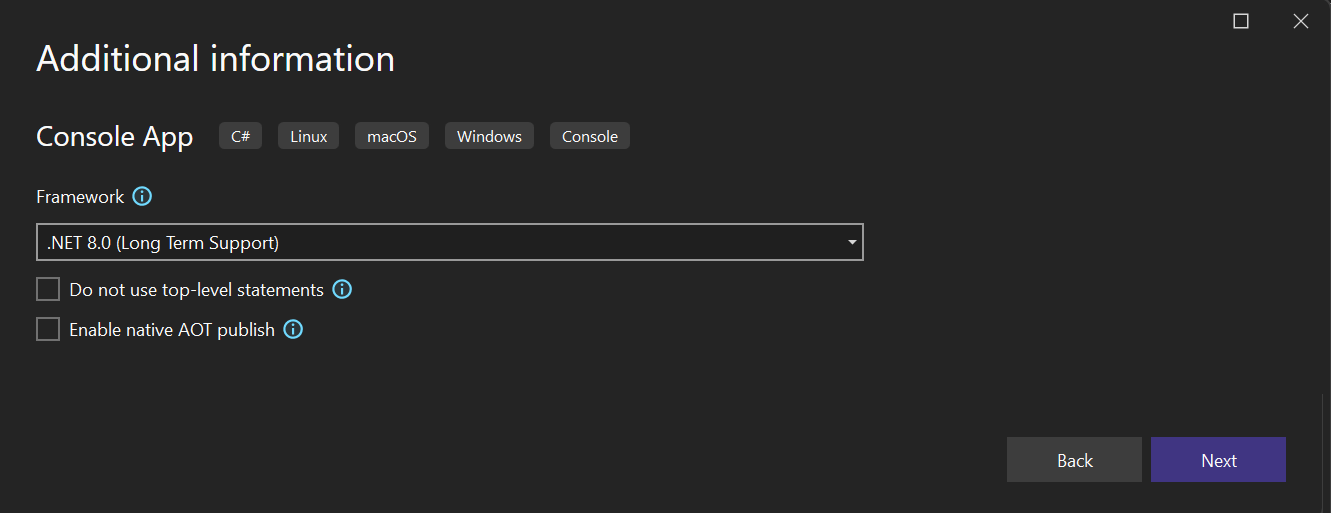
В Visual Studio Installer выберите Изменить. Вам может быть предложено сохранить результаты работы. Нажмите кнопку "Продолжить", чтобы установить рабочую нагрузку.

Вернитесь к шагу 2 в процедуре [Создание проекта](https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/tutorial-console?view=vs-2022#create-a-project).

В окне Настроить новый проект введите *Calculator* в поле Имя проекта, а затем щелкните Далее.

[](https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/media/vs-2022/csharp-name-your-calculator-project.png?view=vs-2022#lightbox)

В окне "Дополнительные сведения" выберите .NET 8.0 для поля Target Framework. Затем выберите Создать.

[](https://learn.microsoft.com/ru-ru/visualstudio/get-started/csharp/media/vs-2022/csharp-target-framework.png?view=vs-2022#lightbox)

Visual Studio открывает новый проект, включающий код по умолчанию "Hello World". Чтобы просмотреть его в редакторе, выберите файл кода *Program.cs* в окне Обозревателя решений, которое обычно находится в правой части Visual Studio.

Один оператор кода вызывает метод [WriteLine](https://learn.microsoft.com/ru-ru/dotnet/api/system.console.writeline) для отображения литеральной строки "Hello, World!" в окне консоли. Если нажать клавишу F5, программа по умолчанию запустится в режиме отладки. После запуска приложения в отладчике окно консоли остается открытым. Для закрытия окна консоли нажмите любую клавишу.

*Система типов C#*

C# является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение. Каждое объявление метода задает имя, тип и вид (значение, ссылка или вывод) для каждого входного параметра и для возвращаемого значения. В библиотеке классов .NET определены встроенные числовые типы и комплексные типы, представляющие разнообразные конструкции. К ним относятся файловая система, сетевые подключения, коллекции и массивы объектов, а также даты. Обычная программа на C# использует типы из этой библиотеки классов и пользовательские типы, которые моделируют уникальные концепции конкретной сферы применения.

В типах может храниться следующая информация:

место, необходимое для хранения переменной этого типа;

максимальное и минимальное значения, которые могут быть представлены;

содержащиеся в типе члены (методы, поля, события и т. д.);

базовый тип, от которого наследует этот тип;

реализуемые им интерфейсы;

разрешенные виды операций.

Компилятор использует сведения о типах, чтобы проверить, все ли операции, выполняемые в коде, являются *типобезопасными*. Например, при объявлении переменной типа [int](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) компилятор позволяет в дополнение использовать переменную и операции вычитания. При попытке выполнить эти же операции для переменной типа [bool](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/bool) компилятор выдаст ошибку, как показано в следующем примере:

C#Копировать

int a = 5;

int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.

int c = a + test;

 Примечание

Тем, кто ранее использовал C и C++, нужно обратить внимание на то, что в C# тип bool нельзя преобразовать в int.

Компилятор внедряет сведения о типе в исполняемый файл в виде метаданных. Среда CLR использует эти метаданные во время выполнения для дальнейшего обеспечения безопасности типа при выделении и освобождении памяти.

Задание типов в объявлениях переменных

Когда вы объявляете в программе переменную или константу, для нее нужно задать тип либо использовать ключевое слово [var](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/var), чтобы компилятор определил тип самостоятельно. В следующем примере показаны некоторые объявления переменных, использующие встроенные числовые типы и сложные пользовательские типы:

C#Копировать

// Declaration only:

float temperature;

string name;

MyClass myClass;

// Declaration with initializers (four examples):

char firstLetter = 'C';

var limit = 3;

int[] source = { 0, 1, 2, 3, 4, 5 };

var query = from item in source

where item <= limit

select item;

Типы параметров и возвращаемых значений метода задаются в объявлении метода. Далее представлена сигнатура метода, который требует значение int в качестве входного аргумента и возвращает строку:

C#Копировать

public string GetName(int ID)

{

if (ID < names.Length)

return names[ID];

else

return String.Empty;

}

private string[] names = { "Spencer", "Sally", "Doug" };

После объявления переменной вы не можете повторно объявить ее с новым типом и назначить ей значение, несовместимое с объявленным типом. Например, нельзя объявить переменную типа int и затем присвоить ей логическое значение true. Но значения можно преобразовать в другие типы, например при сохранении в других переменных или передаче в качестве аргументов метода. Если *преобразование типов* не приводит к потере данных, оно выполняется компилятором автоматически. Для преобразования, которое может привести к потере данных, необходимо выполнить *приведение* в исходном коде.

Дополнительные сведения см. в разделе [Приведение и преобразование типов](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/casting-and-type-conversions).

Встроенные типы

C# предоставляет стандартный набор встроенных типов. Они используются для представления целых чисел, значений с плавающей запятой, логических выражений, текстовых символов, десятичных значений и других типов данных. Также существуют встроенные типы string и object. Такие типы доступны для использования в любой программе C#. Полный список встроенных типов см. в разделе [Встроенные типы](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/built-in-types).

Пользовательские типы

Для создания собственных пользовательских типов используется [struct](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct)[enum](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/enum)[class](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class)[interface](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/interface), и [record](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/record) конструкции , а также конструкции. Сама библиотека классов .NET — это коллекция пользовательских типов, которые вы можете свободно использовать в приложениях. По умолчанию в любой программе C# доступны наиболее часто используемые типы из библиотеки классов. Чтобы сделать доступными другие типы, нужно явным образом добавить в проект ссылку на сборку, которая определяет их. Если компилятору предоставлена ссылка на сборку, то вы можете объявлять в коде переменные (и константы) любых типов, объявленных в этой сборке. См. дополнительные сведения о [библиотеке классов .NET](https://learn.microsoft.com/ru-ru/dotnet/standard/class-library-overview).

*Общие сведения о классах*

Типы ссылок

Тип, определенный *как*[*class*](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class)*ссылочный тип*. Во время выполнения при объявлении переменной ссылочного типа переменная содержит значение [null](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/null) , пока явно не создадите экземпляр класса с помощью [new](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/new-operator) оператора или назначьте его объект совместимого типа, который мог быть создан в другом месте, как показано в следующем примере:

C#Копировать

//Declaring an object of type MyClass.

MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.

MyClass mc2 = mc;

При создании объекта выделяется достаточный объем памяти для этого объекта в управляемой куче, и переменная хранит только ссылку на расположение данного объекта. Память, используемая объектом, освобождается функцией автоматического управления памятью среды CLR, которая называется *сборкой* мусора. Дополнительные сведения о сборке мусора см. в разделе [Автоматическое управление памятью и сборка мусора](https://learn.microsoft.com/ru-ru/dotnet/standard/garbage-collection/fundamentals).

Объявление классов

Классы объявляются с помощью class ключевое слово и уникального идентификатора, как показано в следующем примере:

C#Копировать

//[access modifier] - [class] - [identifier]

public class Customer

{

// Fields, properties, methods and events go here...

}

Дополнительный модификатор доступа предшествует class ключевое слово. Так как [public](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/public) используется в этом случае, любой пользователь может создавать экземпляры этого класса. За именем класса следует ключевое слово class. Имя класса должно быть допустимым [именем идентификатора](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/coding-style/identifier-names) C#. Оставшаяся часть определения — это тело класса, в котором задаются данные и поведение. Поля, свойства, методы и события в классе собирательно называются *членами класса*.

Создание объектов

Хотя иногда они используются взаимозаменяемо, класс и объект отличаются. Класс определяет тип объекта, но это не сам объект. Объект — это конкретная сущность, основанная на классе, которую иногда называют экземпляром класса.

Объекты можно создать с помощью new ключевое слово, за которым следует имя класса, как показано ниже.

C#Копировать

Customer object1 = new Customer();

При создании экземпляра класса ссылка на объект передается программисту. В предыдущем примере object1 представляет собой ссылку на объект, который основан на Customer. Эта ссылка ссылается на новый объект, но не содержит сами данные объекта. Фактически, можно создать ссылку на объект без создания собственно объекта:

C#Копировать

Customer object2;

Мы не рекомендуем создавать ссылки на объекты, которые не ссылаются на объект, так как попытка доступа к объекту через такую ссылку завершается сбоем во время выполнения. Ссылку можно сделать для ссылки на объект, создав новый объект или назначив его существующему объекту, например:

C#Копировать

Customer object3 = new Customer();

Customer object4 = object3;

В этом коде создаются две ссылки на объект, которые указывают на один и тот же объект. Таким образом, любые изменения объекта, выполненные посредством object3, отражаются при последующем использовании object4. Поскольку на объекты, основанные на классах, указывают ссылки, классы называют ссылочными типами.

Конструкторы и инициализация

В предыдущих разделах представлен синтаксис для объявления типа класса и создания экземпляра этого типа. При создании экземпляра типа необходимо убедиться, что его поля и свойства инициализированы в полезные значения. Существует несколько способов инициализации значений:

Примите значения по умолчанию

Инициализаторы полей

Параметры конструктора

Инициализаторы объектов

Каждый тип .NET имеет значение по умолчанию. Как правило, это значение равно 0 для типов чисел и null для всех ссылочных типов. Вы можете полагаться на это значение по умолчанию, если оно разумно в приложении.

Если значение по умолчанию для .NET не является правильным, можно задать начальное значение с помощью инициализатора полей:

C#Копировать

public class Container

{

// Initialize capacity field to a default value of 10:

private int \_capacity = 10;

}

Для предоставления начального значения можно требовать, определив *конструктор* , отвечающий за настройку этого начального значения:

C#Копировать

public class Container

{

private int \_capacity;

public Container(int capacity) => \_capacity = capacity;

}

Начиная с C# 12, можно определить основной *конструктор* в рамках объявления класса:

C#Копировать

public class Container(int capacity)

{

private int \_capacity = capacity;

}

Добавление параметров в имя класса определяет основной *конструктор*. Эти параметры доступны в теле класса, который включает его элементы. Их можно использовать для инициализации полей или в любом другом месте, где они необходимы.

Можно также использовать модификатор для свойства и разрешить вызывающим пользователям использовать required*инициализатор* объектов, чтобы задать начальное значение свойства:

C#Копировать

public class Person

{

public required string LastName { get; set; }

public required string FirstName { get; set; }

}

Добавление required ключевое слово мандатов, которые вызывающие должны задать эти свойства как часть new выражения:

C#Копировать

var p1 = new Person(); // Error! Required properties not set

var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };

Наследование классов

Классы полностью поддерживают *наследование*, фундаментальный механизм объектно ориентированного программирования. При создании класса можно наследовать от любого другого класса, который не определен как [sealed](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed). Другие классы могут наследоваться от класса и переопределять виртуальные методы класса. Кроме того, можно реализовать один или несколько интерфейсов.

При наследовании создается *производный* класс, то есть класс объявляется с помощью *базового класса*, от которого он наследует данные и поведение. Базовый класс задается добавлением после имени производного класса двоеточия и имени базового класса, как показано далее:

C#Копировать

public class Manager : Employee

{

// Employee fields, properties, methods and events are inherited

// New Manager fields, properties, methods and events go here...

}

Если объявление класса включает базовый класс, он наследует все члены базового класса, кроме конструкторов. Дополнительные сведения см. в разделе [Наследование](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/object-oriented/inheritance).

Класс в C# может непосредственно наследоваться только от одного базового класса. Однако, поскольку базовый класс может наследовать от другого класса, класс может косвенно наследовать несколько базовых классов. Кроме того, класс может напрямую реализовать несколько интерфейсов. Дополнительные сведения см. в статье [Интерфейсы](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/interfaces).

Класс можно объявить как [abstract](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/abstract). Абстрактный класс содержит абстрактные методы, которые имеют определение сигнатуры, но не имеют реализации. Не удается создать экземпляр абстрактных классов. Они могут использоваться только через производные классы, реализующие абстрактные методы. В отличие от этого, запечатанный класс не позволяет другим классам быть производным от него. Дополнительные сведения см. в статье [Абстрактные и запечатанные классы и члены классов](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members).

Определения классов можно разделить между различными исходными файлами. Дополнительные сведения см. в разделе [Разделяемые классы и методы](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods).

*Тестирование в .NET*

В этой статье рассматривается концепция тестирования и демонстрируется использование разных тестов для проверки кода. Для тестирования приложений .NET существует много средств, например [.NET CLI](https://learn.microsoft.com/ru-ru/dotnet/core/testing/#net-cli) и [интегрированные среды разработки (IDE)](https://learn.microsoft.com/ru-ru/dotnet/core/testing/#ide).

Типы тестов

Наличие автоматических тестов — отличный способ убедиться, что код приложения работает именно так, как задумывалось разработчиками. В этой статье рассматриваются модульные тесты, интеграционные тесты и нагрузочные тесты.

Модульные тесты

Модульный тест позволяет проверить отдельные программные компоненты или методы, также именуемые "единицей работы". Модульные тесты должны проверять только тот код, который может изменять конкретный разработчик. В них не затрагиваются аспекты инфраструктуры. К инфраструктуре здесь относятся любые взаимодействия с базами данных, файловыми системами и сетевыми ресурсами.

Дополнительные сведения о создании модульных тестов см. в документации по [средствам тестирования](https://learn.microsoft.com/ru-ru/dotnet/core/testing/#testing-tools).

Интеграционные тесты

Тест интеграции отличается от модульного теста тем, что он позволяет двум или более программным компонентам работать совместно, также называемый интеграцией. Эти тесты работают в более широком спектре тестируемой системы, в то время как модульные тесты сосредоточены на отдельных компонентах. Интеграционные тесты часто содержат некоторые элементы взаимодействия с инфраструктурой.

Нагрузочные тесты

Нагрузочный тест определяет, может ли система справиться с заданным уровнем нагрузки, например выдержать определенное число одновременных подключений пользователей к приложению и быстро отвечать на запросы взаимодействия. Дополнительные сведения о нагрузочном тестировании веб-приложений см. в статье [Нагрузочное тестирование в ASP.NET Core](https://learn.microsoft.com/ru-ru/aspnet/core/test/load-tests).

Рекомендации по тестированию

Не забывайте, что для написания тестов есть [рекомендации](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-best-practices). Например, метод [разработки на основе тестирования](https://deviq.com/test-driven-development) подразумевает, что модульный тест всегда пишется раньше того кода, который он проверяет. Разработка на основе тестирования аналогична созданию оглавления книги до того, как вы начнете ее писать. Она помогает разработчикам писать более простой, удобочитаемый и эффективный код.

Средства тестирования

Платформа .NET поддерживает разработку на разных языках программирования и создание тестов разных типов для [C#](https://learn.microsoft.com/ru-ru/dotnet/csharp/), [F#](https://learn.microsoft.com/ru-ru/dotnet/fsharp/) и [Visual Basic](https://learn.microsoft.com/ru-ru/dotnet/visual-basic/). Для каждого из этих языков можно выбрать любую из нескольких платформ тестирования.

xUnit

[xUnit](https://xunit.net/) — это бесплатное средство модульного тестирования для .NET с открытым кодом и широкой поддержкой сообщества. Это новейшая технология для модульного тестирования приложений .NET, созданная первым разработчиком NUnit v2. xUnit.net поддерживает ReSharper, CodeRush, TestDriven.NET и [Xamarin](https://dotnet.microsoft.com/apps/xamarin). Это проект [.NET Foundation](https://dotnetfoundation.org/), поэтому он работает в соответствии с их правилами поведения.

Дополнительные сведения см. в следующих ресурсах:

[Модульное тестирование в C#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-with-dotnet-test)

[Модульное тестирование в F#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-fsharp-with-dotnet-test)

[Модульное тестирование в Visual Basic](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-visual-basic-with-dotnet-test)

NUnit

[NUnit](https://nunit.org/) — это платформа модульного тестирования для всех языков .NET. Изначально она была перенесена из JUnit, а текущий рабочий выпуск основательно переработан с добавлением новых функций и широкого спектра поддерживаемых платформ .NET. Это проект [.NET Foundation](https://dotnetfoundation.org/).

Дополнительные сведения см. в следующих ресурсах:

[Модульное тестирование в C#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-with-nunit)

[Модульное тестирование в F#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-fsharp-with-nunit)

[Модульное тестирование в Visual Basic](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-visual-basic-with-nunit)

MSTest

[MSTest](https://github.com/Microsoft/testfx-docs) — это тестовая платформа корпорации Майкрософт для всех языков .NET. Она поддерживает расширение и работу с .NET CLI и Visual Studio. Дополнительные сведения см. в следующих ресурсах:

[Модульное тестирование в C#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-with-mstest)

[Модульное тестирование в F#](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-fsharp-with-mstest)

[Модульное тестирование в Visual Basic](https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-visual-basic-with-mstest)

Интерфейс командной строки.NET

Модульные тесты для решений можно выполнять в [.NET CLI](https://learn.microsoft.com/ru-ru/dotnet/core/tools/) с помощью команды [dotnet test](https://learn.microsoft.com/ru-ru/dotnet/core/tools/dotnet-test). .NET CLI предоставляет почти все функциональные возможности, доступные через пользовательские интерфейсы [интегрированных сред разработки](https://learn.microsoft.com/ru-ru/dotnet/core/testing/#ide). .NET CLI работает на многих платформах и успешно встраивается в конвейеры непрерывной интеграции и доставки. .NET CLI используется в скриптах процессов, выполняющих автоматизацию типичных задач.

IDE

Независимо от того, используете ли вы Visual Studio, Visual Studio для Mac или Visual Studio Code, вам будут доступны графические пользовательские интерфейсы для тестирования функций. Для сред IDE предоставляется больше возможностей, чем доступно в CLI, например [Live Unit Testing](https://learn.microsoft.com/ru-ru/visualstudio/test/live-unit-testing). Дополнительные сведения см. в разделе о [добавлении и исключении тестов в Visual Studio](https://learn.microsoft.com/ru-ru/visualstudio/test/live-unit-testing#include-and-exclude-test-projects-and-test-methods).

*Дорожная карта*1. Общее представление о разработке и Web

**а) Понять, как работает веб.**

C# : <https://dotnet.microsoft.com/learn/csharp>

Основы Linux: <https://stepik.org/course/762/promo>

Web Dev (HTML and CSS: <https://freecodecamp.org/learn/2022/responsive-web-design/>

**Введение в информатику 101** от Стендфорда: <https://online.stanford.edu/courses/soe-ycscs101-computer-science-101>

б) Изучить инструменты контроля версий

Git

GitHub

GitLab (полезно знать)

**Курсы:**

Основы Git: Курс описывает наиболее часто используемые команды git, взятые из опыта реальных проектов. Даются примеры и интерактивные задачи : <https://stepik.org/course/3145/promo>

Git и GitHub бесплатный курс поможет вам разобраться с контролем версий: [http://learngitbranching.js.org](http://learngitbranching.js.org/)

Git введение: <http://mygreatlearning.com/academy/learn-for-free/courses/github-tutorial-for-beginners>

в) Научитесь искать решения в сети.

Google

Stack Overflow

Bing Search

ChatGPT

д) Изучите Структуры данных и алгоритмы ( полезные бесплатные курсы для изучения стуртур данных можно найти [тут](https://uproger.com/1000-besplatnyh-onlajn-kursov-po-programmirovaniyu-aktualnye-v-2023-godu/))

**2 Изучение С#**

Бесплатные курсы для изучения C#:

[Изучите базовый синтаксис и процессы, необходимые для создания простых приложений на языке C#](https://learn.microsoft.com/ru-ru/training/paths/get-started-c-sharp-part-1/)

[Basics of Object-Oriented Programming with C#](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fbasics-of-object-oriented-programming-with-csharp%2F)

[Fundamentals of Programming: Understanding C#](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Funderstandingc%2F)

[C# Fundamentals](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fcsharp-fundamentals-csharp5)

b) Изучите .NET 8

[Курс: Создание приложений .NET на C#](https://learn.microsoft.com/ru-ru/training/paths/build-dotnet-applications-csharp/)

c) .NET CLI

[Обзор интерфейса командной строки .NET](https://learn.microsoft.com/ru-ru/dotnet/core/tools/)

3 Принципы SOLID

[**SOLID**](https://habr.com/ru/articles/688530/) – это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

Узучите:  
a) Принцип единой ответственности

б) Принцип открытости-закрытости

c) Принцип замещения Лискова (хорошо знать)

г) Принцип разделения интерфейсов

д) Принцип инверсии зависимостей

f) DRY

g) YAGNI

h) KISS

[SOLID ПРИНЦИПЫ простым языком (много примеров)](https://www.youtube.com/watch?v=TxZwqVTaCmA)

4 Изучите основы ASP.NET Core

Вот основные темы для изучения:

а) Веб-интерфейсы API

б) Минимальные APIe

в) Маршрутизация

г) Мидлвар

д) Фильтры и атрибуты

f) Конфигурация

g) Аутентификация и авторизация

h) Инъекция зависимостей

a) Основы РСУБД и проектирование баз данных

б) Синтаксис SQL

c) Хранимые процедуры (полезно знать)

Курсы:

[Создание веб-приложений с помощью ASP.NET Core для начинающих](https://learn.microsoft.com/ru-ru/training/paths/aspnet-core-web-app/)

[Создание облачных приложений и служб с помощью .NET и ASP.NET Core](https://learn.microsoft.com/ru-ru/training/paths/create-microservices-with-dotnet/)

[Структура ASP.NET проекта: Все, что вам нужно знать](https://www.youtube.com/watch?v=O6JgBnGMqJo)

​🌟 **Шаблоны архитектуры приложений**

Прежде чем приступить к разработке программного обеспечения, мы должны выбрать подходящую архитектуру, которая обеспечит нам желаемую функциональность и качественные характеристики.

Статья рассказывает об основных шаблонах архитектуры приложений:

<https://www.redhat.com/architect/14-software-architecture-patterns>

**г) Базы данных**

Postgres

SQL Server

MySQL/MariaDB (хорошо знать)

**–**[**Полный учебный план по изучению баз данных**](https://t.me/sqlhub/786)

План рассчитан на 28 дней, в течение которых необходимо уделять не менее 1,5 часов в день**.**

– [**SQL дорожная карта 2024 года**](https://uproger.com/sql-dorozhnaya-karta-2024-goda/)

5 Работа с API

а) REST

**Web**

Минимальное API

FastEndpoints (полезно знать)

[DevDocs](https://devdocs.io/go/) **— все API-документации в одном месте**

Главные преимущества DevDocs включают объединение множества официальных документаций по разработке, удобный поиск и быстрый доступ к информации.

Особенно полезным инструмент будет для студентов IT-направлений, но свою пользу смогут найти в нём и работающие специалисты

Из дополнительных плюсов: интуитивно понятный интерфейс, настраиваемые закладки и автодополнение поиска

б) GraphQL (полезно знать)

[Введение в GraphQL](https://highload.today/graphql/)

в) gRPC (полезно знать)

Курсы:

[**Подготовка к сертификации ISTQB Foundation**](https://stepik.org/course/16478/syllabus)

[Postman для тестирования APIС](https://stepik.org/course/120679/info)

[**Тестирование ПО: подготовка к сертификации ISTQB Foundation**](https://stepik.org/course/16478/syllabus)

6 ORM

[Объектно-реляционное отображение (ORM)](https://ru.wikipedia.org/wiki/ORM)  
а) Dapper

b) Entity Framework Core

Изучите основы EF Core

Миграции

API отслеживания изменений

Ленивая загрузка, ускоренная загрузка, явная загрузка

[**Как использовать ORM в .NET Core и C#.**](https://softchris.github.io/pages/dotnet-orm.html)

7 **Внедрение зависимостей в .NET**

**Внедрение зависимостей в .NET** — это встроенная часть платформы, а также конфигурация, ведение журнала и шаблон параметров.

**Внедрение зависимостей в .NET** – это подход, который позволяет классам получать свои зависимости из внешнего источника, вместо того чтобы создавать их самостоятельно. **Внедрение зависимостей**позволяет отделить процесс создания объектов от кода, который их использует, что приводит к более модульной и гибкой системе.

Почему это нужно:

Классы отделяются друг от друга, поэтому изменения, внесенные в одно место, отражаются во всех остальных местах.

Три способа внедрения зависимостей:

**▪️Scoped:** Создается экземпляр на каждую область видимости.

**▪️Transient:** Создается новый экземпляр каждый раз при его внедрении.

**▪️Singleton:** Создается один объект для всех запросов, поступающих на сервер от любого пользователя.

**Курс**: [Microsoft.Extensions.DependencyInjection](https://learn.microsoft.com/ru-ru/dotnet/core/extensions/dependency-injection)

b) Scrutor (полезно знать)

a) Облачные базы данных

AWS DynamoDB

Azure CosmosDB

<https://explore.skillbuilder.aws/learn/signin>

б) ElasticSearch

в) [Redis](https://university.redis.com/)

г) [MongoDB](https://t.me/sqlhub/783) (полезно знать)

Docker – <http://cognitiveclass.ai/courses/docker-essentials>

10 Кэширование

a) Кэширование вывода

б) Кэширование ответов (полезно знать)

StackExchange.Redis

11 Ведение логов

a) Microsoft.Extensions.Logging – <https://learn.microsoft.com/en-us/dotnet/core/extensions/logging>

б) Serilog

12 Брокеры сообщений

a) Azure Service Bus

c) RabbitMQ (полезно знать)

b) AWS SQS/SNS

г) MassTransit

[Брокеры сообщений, или Как происходит взаимодействие в рамках распределённой инфраструктуры](https://habr.com/ru/companies/sberbank/articles/669456/)

13 Потоковая передача данных

a) Apache Kafka –

<https://developer.confluent.io/courses/>

(полезно знать)

b) AWS Kinesis (хорошо знать)

c) Azure Event Hubs (хорошо знать)

14 Тестирование

а) модульное тестирование

“Библия QA” – это обновляемая база знаний объемом 560+ страниц:

▪Ответы на самые популярные вопросы новичков о профессии и старте карьеры;

▪Крупнейшая подборка ссылок и полезных ресурсов;

▪Конспект всевозможной теории и ответов на вопросы с реальных собеседований.

<https://github.com/VladislavEremeev/QA_bible>

15 Коммуникация в реальном времени

a) SignalR <https://learn.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-8.0>

b) Web Sockets (полезно знать) – <https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_server>

16 Документация API

a) OpenAPI/Swagger – <https://swagger.io/specification/>

б) AsyncAPI (полезно знать) –<https://www.asyncapi.com/tools>

17 Библиотеки API SDK

а) Refit

б) RestSharp

в) Flurl (хорошо знать)

<https://github.com/microsoftgraph/msgraph-sdk-dotnet>

18 Планирование задач

a) BackgroundService

б) PeriodicTimer

в) HangFire (полезно знать)

<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler?view=net-8.0>

19 Мониторинг и телеметрия

a) OpenTelemetry – <https://t.me/Golang_google/1597>

Jeager (полезно знать)  
б) Prometheus

в) Grafana

г) ELK Stack

д) Datadog (полезно знать)

20 Контейнеры

а) Контейнеризация

Docker – <https://learn.microsoft.com/en-us/dotnet/core/docker/build-container>  
Podman (полезно знать)- <https://developers.redhat.com/articles/2022/03/21/hello-podman-using-net>б) Оркестровка

Kubernetes

21 Облачные технологии

а) Облачные провайдеры – <https://www.youtube.com/watch?v=TCuxXg0AUao>

-Azure  
– AWS  
б) Бессерверные технологии

Azure Functions  
AWS Lambda  
в) Файловое хранилище

AWS S3  
Azure Storage

**Бесплатные Курсы:**

AWS →[http://explore.skillbuilder.aws](http://explore.skillbuilder.aws/)

Azure →[http://learn.microsoft.com](http://learn.microsoft.com/)

DevOps →<http://edx.org/learn/devops>

Docker →[http://docker-curriculum.com](http://docker-curriculum.com/)

Kubernetes →[http://kubernetes.io](http://kubernetes.io/)

22 Непрерывная интеграция и развертывание

a) GitHub Actions

б) TeamCity

c) Octopus Deploy

d) Azure Pipelines (полезно знать)

д) Gitlab CI (полезно знать)

f) Jenkins (хорошо знать)

g) Автоматизация сборки (хорошо знать)

Cake (полезно знать)  
Nuke (хорошо знать)

❯ **Jenkins**

<http://mygreatlearning.com/academy/learn-for-free/courses/jenkins-tutorial>

❯ **Puppet**

<http://simplilearn.com/free-course-to-learn-puppet-basics-skillup>

❯ **Chef**

<http://simplilearn.com/chef-in-devops-basics-free-course-skillup>

❯ **Ansible**

<http://simplilearn.com/learn-ansible-basics-free-course-skillup>

❯ **Selenium**

<http://mygreatlearning.com/academy/learn-for-free/courses/selenium-basics>

❯ **VirtualBox & vSphere**

<http://openclassrooms.com/courses/7163136-set-up-virtual-machines-using-virtualbox-and-vsphere>

23 Библиотеки .NET

a) Polly

b) FluentValidation

c) Humanizer.Core

d) Benchmark.NET

e) MediatR (полезно знать)

f) Units.NET (полезно знать)

g) NodaTime (полезно знать)

[**Awesome-C#**](https://github.com/uhub/awesome-c-sharp)— огромный список полезных фреймворков, готовых скриптов, библиотек и программного обеспечения.

24 DevOps

[**Бесплатный онлайн-курс DevOps**](https://academiait.ru/course/devops/)

б) Terraform (хорошо знать)

в) Pulumi (хорошо знать)

25 Пишите код каждый день”

Поздравляем вас с достижением этого этапа дорожной карты!

Помните, что постоянное обучение и кодинг – это ключ к освоению .NET Framework. Не теряйте любопытства, изучайте новые технологии и продолжайте создавать интересные проекты. Ваш путь разработчика .NET только начинается, а возможности безграничны.