С.А. Петров

Техническое обеспечение информационных систем: введение в разработку на Visual Studio С#

Учебное пособие

по курсам

«Проектирование информационных систем», «Проектный практикум», «Управление разработкой информационных систем» для студентов, обучающихся по направлениям «Прикладная информатика», «Бизнес-информатика».

***Аннотация***

Пособие предназначено для студентов, желающих познакомиться с платформой .NET Framework, языком C# и средой разработки Microsoft Visual Studio.

.NET Framework — это программная платформа, разрабатываемая компанией Microsoft с 2002 года. Включает в себя среду исполнения программ Common Language Runtime и множество библиотек, облегчающих создание приложений. Язык программирования C# является одним из основных языков написания приложений для платформы Microsoft .NET. Он был разработан в 1998-2001 годах и постоянно совершенствуется, впрочем, как и остальные продукты, рассмотренные в данном пособии. Наиболее популярной средой разработки, поддерживающей .NET, является Microsoft Visual Studio. Visual Studio начинает свой путь с 1997 года и позволяет разрабатывать приложения на различных языках программирования, включая C#.

Для каждого продукта год его основания упоминается неспроста. Он показывает, что текущая версия имеет богатую историю и включает в себя множество исправлений, доработок, улучшений и охватывает огромную часть мира разработки программного обеспечения. Все продукты крепко «стоят на ногах». Это, несомненно, важно в эпоху множества программных платформ (фреймворков), которые периодически появляются, привнося что-то новое, и исчезают, забирая с собой все усилия, потраченные на их изучение и использование. Овладение представленными в данном пособии средствами пригодится тому, кто желает связать свою жизнь с разработкой программного обеспечения, а время, потраченное на их изучение, не пропадёт зря.

Материал содержит описание основ программирования на языке C# и показывает практическое применение объектно-ориентированного подхода при реализации консольного меню для выполнения нескольких арифметических операций.

Содержание

[1. Первое приложение 4](#_Toc467485938)

[2. Основы языка C# 8](#_Toc467485939)

[2.1. Базовые типы данных в C# 8](#_Toc467485940)

[2.2. Базовые конструкции в C# 10](#_Toc467485941)

[2.3. Контрольные вопросы и задания 15](#_Toc467485942)

[3. Основы объектно-ориентированной разработки на языке C# 16](#_Toc467485943)

[3.1. Понятие класса 16](#_Toc467485944)

[3.2. Инкапсуляция 17](#_Toc467485945)

[3.3. Наследование 18](#_Toc467485946)

[3.4. Полиморфизм 19](#_Toc467485947)

[3.5. Абстракция 20](#_Toc467485948)

[3.6. Контрольные вопросы и задания 21](#_Toc467485949)

[4. Основы работы в Visual Studio 21](#_Toc467485950)

[4.1. Работа со структурой решения 22](#_Toc467485951)

[4.2. Сборка и запуск решения 23](#_Toc467485952)

[4.3. Контрольные вопросы и задания 24](#_Toc467485953)

[5. Разработка консольного меню 25](#_Toc467485954)

[5.1. Проектирование 25](#_Toc467485955)

[5.2. Разработка 27](#_Toc467485956)

[5.3. Пример использования 29](#_Toc467485957)

[5.4. Контрольные вопросы и задания 31](#_Toc467485958)

[Заключение 31](#_Toc467485959)

[Библиографический список 32](#_Toc467485960)

# Первое приложение

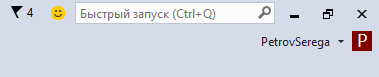
В данном пособии для разработки компьютерных программ будет использоваться среда разработки Microsoft Visual Studio. Средой разработки или интегрированной средой разработки называется комплекс программных средств, применяемый для создания программного обеспечения.

Microsoft предлагает различные редакции Visual Studio, начиная с бесплатной версии Visual Studio Community и заканчивая требующей серьёзные финансовые вложения Visual Studio Enterprise (на конец 2016 года — 250$ в месяц за одно рабочее место). Редакции различаются предлагаемым пользователю функционалом. Примеры, разобранные в данном пособии, можно повторить, используя бесплатную редакцию Community, загрузка которой доступна с официального сайта: <https://www.visualstudio.com/> [1].

При установке следует оставить все значения параметров по умолчанию. Для запуска установленной среды используйте ярлык на рабочем столе или соответствующий пункт меню «Пуск». После успешного запуска Visual Studio откроется начальная страница. На ней отображается справочная информация о продукте, новости, описание новых возможностей, а также меню для создания нового решения или открытия существующего.

Для работы с редакцией Community необходимо осуществить вход с помощью учётной записи Microsoft Account. Для этого надо нажать на кнопку «Вход» в правом верхнем углу и в появившемся окне ввести данные своей учётной записи. В случае отсутствия учетной записи, необходимо создать её (<https://login.live.com/RU>, [2]), иначе Visual Studio Community будет недоступна после окончания «триального» срока использования.

После успешно осуществленного входа в левом верхнем углу вместо кнопки «Вход» отобразится имя пользователя (см. рис. 1.1).

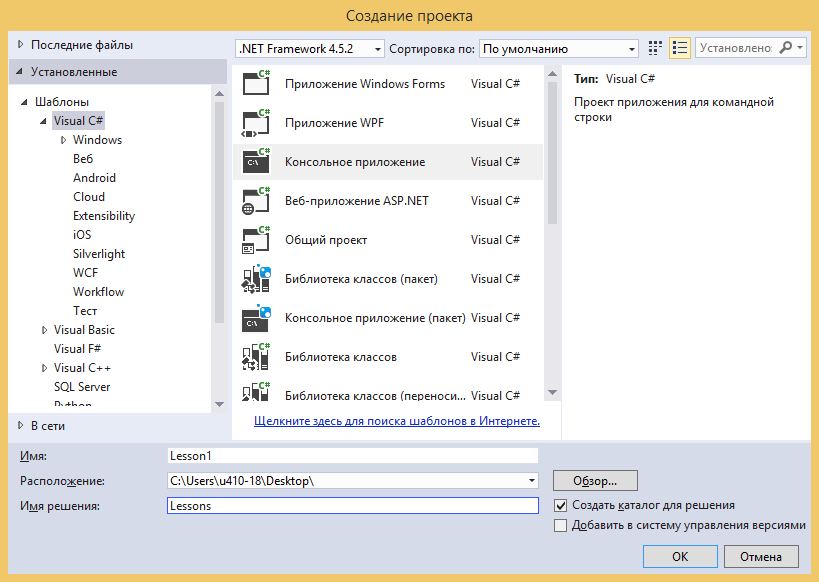


**Рис. 1.1. Вход успешно осуществлён**

Среда разработки Microsoft Visual Studio (MSVS) поддерживает большое количество различных типов проектов. От простейших локальных приложений с минимальным графическим интерфейсом для персональных компьютеров до кроссплатформенных сетевых игр, использующих всю мощность современных компьютеров. MSVS позволяет создавать сервисы и веб-приложения, доступные по сети, приложения для мобильных платформ, решения для баз данных и многое другое.

Одним из наиболее простых, с точки зрения понимания, проектов является проект консольного приложения. В приложении этого типа легко сделать акцент на логике работы, а не на графическом оформлении, которое может усложнить начальное знакомство с устройством программы.

Для создания нового приложения необходимо выбрать в верхнем горизонтальном меню раздел «Файл», далее: «Создать → Проект». Отобразится окно для выбора типа приложения:



**Рис. 1.2. Выбор типа нового приложения**

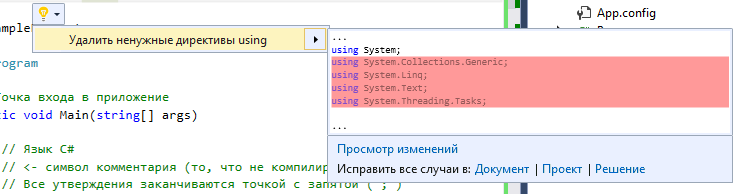
В окне создания проекта отображены поддерживаемые типы приложений. Visual Studio поддерживает установку дополнительных шаблонов приложений из сети. Для создания консольного приложения на языке C# необходимо в разделе «Visual C#» (проекты, разрабатываемые на языке C#) выбрать «Консольное приложение». При создании нового проекта следует указать имя приложения (проекта), его расположение, имя всего решения (решение состоит из нескольких проектов), создавать ли отдельный каталог для решения, следует ли добавлять проект в систему управления версиями (например, в Team Foundation Server). После задания всех параметров необходимо нажать на кнопку «ОК».

Процесс создания нового приложения занимает определённое время. Ход этого процесса отображается в левой нижней области окна в строке состояния. В данной строке должны отобразиться этапы «Создание проекта» и «Проект успешно создан». В ходе разработки приложения исходные файлы часто редактируются и требуют сохранения. Профессиональные разработчики для сохранения используют сочетание клавиш «Ctrl + S» (сохранить текущий файл) или «Ctrl + Shift + S» (сохранить все файлы решения). При успешном сохранении файла в строке состояния появится сообщение «Элементы сохранены».

После создания проекта его структура доступна для просмотра в обозревателе решений. Созданный проект содержит директорию «Свойства» для указания свойств сборки, раздел «Ссылки», описывающий задействованные в проекте сборки, файл конфигурации приложения App.config и файл Program.cs, содержащий точку входа в программу — функцию Main.

Файл Program.cs уже содержит исходный код. Обычно каждый отдельный файл программы на языке C# начинается с подключения необходимых пространств имён. Файл Program.cs не исключение. Пространства имён (namespace) применяются для логической группировки классов, обеспечивающих определённую функциональность. Для работы с классом необходимо подключить пространство имён, в котором он определён. Для этого служит ключевое слово using, после которого указывается конкретное пространство имён. В Program.cs уже добавлены основные пространства имен для консольного приложения. Часть из них имеют прозрачный цвет шрифта, подсказывающий, что пока классы из этого пространства имён не задействованы.

После того как программа начнёт обретать функциональность, для некоторых из using-ов прозрачность исчезнет. Остальные можно удалить вручную, либо с помощью средств рефакторинга, входящих в состав MSVS 15. Например, нажать на «фонарик», который появится при наведении курсора на нужную строку, и выбрать пункт «Удалить ненужные директивы using» (см. рис. 1.3). Рефакторинг (от англ. refactoring) или реорганизация кода  — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.



**Рис. 1.3. Удаление неиспользуемых пространств имён**

Исторически сложилось в начале освоения языка программирования выводить на экран приветственное сообщение «Hello, world!» и «Привет, мир!». Не будем обходить эту традицию. Язык C# — это объектно-ориентированный язык, который предполагает описание программы в терминах классов и объектов (экземпляров класса). Не углубляясь в реализацию классов в конкретном языке программирования, можно сказать, что класс — это описание множества объектов с одинаковыми атрибутами, операциями (методами), связями и семантикой [3].

Обычный класс поддерживает создание экземпляров, и для доступа к методам и полям класса эти экземпляры должны быть созданы и размещены в памяти. Но если в описание метода добавить ключевое слово static, то его можно будет вызывать без создания экземпляра класса. Использование пометки static для всего класса показывает, что класс содержит только статические методы и поля. Пример статического класса — класс Console, который предоставляет методы для работы с консольным окном, а пример статического метода — его метод Write, позволяющий выводить строку на экран. Разновидностью метода Write является метод WriteLine, который после вывода информации на экран осуществляет переход на новую строку. Программа, выводящая на экран сообщение «Hello, world!», состоит всего из двух строк:

Console.WriteLine("Hello, world");

Console.ReadKey();

Статический метод ReadKey необходим, чтобы окно программы не закрывалось сразу после вывода приветственного сообщения, и пользователь успел его заметить. В случае его использования программа будет ожидать нажатия пользователем любой клавиши, и только после этого продолжит выполнение. Результатом работы метода ReadKey является информация о нажатой клавише. В данном примере эта информация нас не интересует.

После начала использования методов класса Console строка «using System» перестала быть прозрачной, так как класс Console находится в пространстве имён System.

Для запуска программы необходимо использовать кнопку «Пуск» (Start) в верхней части окна MSVS:



**Рис. 1.4. Кнопка «Пуск» для старта программы**

В результате работы программы появится черное окно консоли с надписью «Hello, world». После нажатия любой клавиши окно исчезнет.

Класс Console содержит поля и методы для работы с консольным окном. Для примера изменим фон окна и цвет шрифта так, чтобы вывод осуществлялся «чёрным по белому». Для этого изменим значение двух статических полей класса Console до начала вывода информации на экран:

// Изменение цвета фона

Console.BackgroundColor = ConsoleColor.White;

// Изменение цвета шрифта

Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine("Hello, world");

Console.ReadKey();

Свойство BackgroundColor отвечает за цвет фона, ForegroundColor — за цвет шрифта. Запустите программу, чтобы убедиться, что фон окна и цвет шрифта изменились.

В листинге выше использованы две косые черты «//» для комментирования участка кода, т. е. указания того, что он не относится к исходному коду программы. Две косые черты служат для комментирования одной строки. Большее количество строк можно закомментировать, заключив их внутрь символов «/\*» и «\*/» (этот подход обычно используется при отладке компьютерной программы).

# Основы языка C#

## Базовые типы данных в C#

Итак, мы разработали простейшую программу на языке программирования C#. Теперь познакомимся с понятием переменная и базовыми типами данных, без которых не обходится создание любой значимой программы.

Переменная — это область в памяти, к которой мы можем обращаться по имени для записи или чтения данных. Данные, находящиеся в памяти, на которую указывает переменная, называются значением переменной. Область памяти описывается типом переменной. Тип данных (тип переменной) — это множество значений. В некоторых случаях с типом связывают множество допустимых операций. Наиболее используемые типы в языке C#:

* bool (System.Boolean) — множество, состоящее из двух значений: правда (true) и ложь (false);
* int (System.Int32) — множество целочисленных чисел;
* string (System.String) — множество строк;
* double (System.Double) — множество чисел с плавающей точкой;
* char (System.Char) — множество символов.

До скобок указано сокращение, которое используется для обозначения соответствующего типа. В скобках — полный путь к имени типа с пространством имён, в котором он располагается.

Тип используется в контексте переменной и позволяет указать какое множество значений может принимать переменная. Например, переменная типа bool может принимать значение true или false, а переменная типа int — значения в диапазоне от (-2^31) до (2^31 — 1). По типу данных можно определить, сколько места занимает переменная в оперативной памяти компьютера. Для хранения переменной типа bool нужен 1 бит (одна ячейка памяти), а для переменной типа int — 32 бита.

Для объявления переменных используется следующий синтаксис:

*<тип> <имя переменной>*

, где типом является любой из вариантов: сокращение, полный путь к имени типа, указание имени типа (с учётом наличия в файле подключения соответствующего пространства имён). В качестве имён переменных используются комбинации латинских символов, но также можно использовать цифры и кириллицу. Пример объявления переменных:

int intVar;

double doubleVar;

String stringVar;

Char charVar;

System.Boolean BooleanVar;

Важным свойством современных программ является интерактивность — взаимодействие с пользователем для совместного решения задачи. Пользователь может задавать исходные данные или выбирать путь решения. Реализовать интерактивность в консольном приложении можно следующим образом:

string userInput = Console.ReadLine();

Для сохранения значения, введённого пользователем, объявляется переменная userInput. String —тип переменной userInput. Само значение получается с помощью вызова метода ReadLine. Он похож на знакомый нам метод ReadKey, за исключением того, что позволяет ввести строку и ожидает нажатия на клавишу Enter. После соответствующих действий пользователя программа продолжает своё выполнение, а переменная userInput содержит значение, указанное пользователем.

Переменные одного типа могут быть объединены в массив. Такой подход хорош, если переменные логически связаны между собой. Например, программный код, позволяющий сохранить 3 строки, введённые пользователем, выглядит так:

// Инициализация строкового массива размером 3

string[] userInput = new string[3];

// Присвоение значения первому элементу массива

userInput[0] = Console.ReadLine();

// Присвоение значения второму элементу массива

userInput[1] = Console.ReadLine();

// Присвоение значения третьему элементу массива

userInput[2] = Console.ReadLine();

При объявлении переменной userInput в левой части используются квадратные скобки для указания того, что переменная является массивом. В правой части в квадратных скобках задаётся размер (количество элементов) массива. После инициализации массива можно обращаться к отдельным элементам, указывая в квадратных скобках номер (индекс) элемента. Нумерация начинается с нуля. Если заранее количество элементов множества не задано, то вместо массива удобнее использовать список, а для добавления элементов – метод Add:

// Инициализация списка строк

List<string> userInput = new List<string>();

// Добавление строки в список

userInput.Add(Console.ReadLine());

// Добавление ещё одной строки в список

userInput.Add(Console.ReadLine());

Существуют и другие типы данных для хранения множества элементов. Например, словарь (Dictionary), ориентированный на некоторый ключ для работы с элементами множества. Примером пары ключ-значение являются: номер зачётной книжки ↔ студент, номер автомобиля ↔ владелец, город ↔ количество жителей.

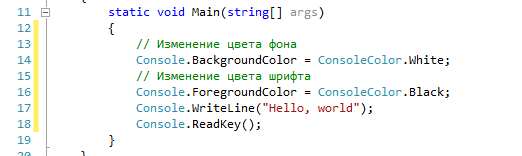
В данном разделе определено понятие переменной, рассмотрен минимальный набор базовых типов данных в C#, синтаксис и семантика работы с ними. Сократить временные и трудовые затраты на разработку нового программного обеспечения позволяет повторное использование кода или готовых модулей. Поэтому, чем выше знание стандартных возможностей платформы разработки (типы данных, вспомогательные методы и т.д.), тем быстрее можно реализовать необходимый функционал. Большое количество технической документации по технологиям Microsoft представлено на сайте Microsoft Developer Network (<https://msdn.microsoft.com/>, [4]). Часть материала интегрирована в Visual Studio и доступна «оффлайн» в разделе «Справка» (Help). Мы же перейдём к рассмотрению механизмов языка C# для описания и контроля хода выполнения программы.

## Базовые конструкции в C#

Если обратиться к определению понятия язык программирования, то там будет сказано, что это формальная знаковая система, предназначенная для записи компьютерных программ. В свою очередь, компьютерная программа — это последовательность [инструкций](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), определяющих процедуру решения конкретной задачи [компьютером](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) (вычислительной машиной). Какими же свойствами должен обладать язык программирования, чтобы на нём было возможно описать компьютерную программу?

На этот вопрос отвечает теорема Бёма-Якопини, которая гласит: «любой исполняемый алгоритм (процедура решения конкретной задачи) может быть преобразован к структурированному виду, т.е. такому виду, когда ход его выполнения определяется только при помощи трёх [структур](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0) управления: последовательной, [ветвлений](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%B2%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) и повторов или [циклов](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)». Эти три структуры управления и должен поддерживать язык программирования для описания любого алгоритма.

Последовательность в языке C# осуществляется за счёт выполнения инструкций в порядке их записи. Инструкции разделяются символом «;» (точка с запятой) и обычно каждая новая инструкция начинается с новой строки. В среде разработки Visual Studio по умолчанию не отображаются номера строк в листинге программы. Чтобы включить их отображение необходимо зайти в «Сервис → Параметры → Текстовый редактор → C#» и отметить галку «Номера строк». После этих действий слева от исходного кода будут отображены номера строк:



**Рис. 2.1. Отображение номеров строк в редакторе Visual Studio**

В данном случае первой выполняется инструкция на 14-ой строке, далее выполняется инструкция на 16-ой строке, так как 15-ая строка закомментирована и не относится к исполняемой части программы, и т.д.

Перед изучением ветвлений и циклов обратимся к оператору, который напрямую не связан с контролем потока выполнения программы, но используется практически в любой компьютерной программе. Это оператор присваивания значения переменной. Мы уже использовали его, когда изменяли цвет фона и шрифта консольного окна.

Синтаксис простой — в левой части инструкции необходимо указать переменную, которой присваивается значение, затем символ «=» и присваиваемое значение:

*<переменная для присваивания значения> = <новое значение переменной>*

Присваивание значения переменной является важной частью написания любой программы, так как инструкции обычно имеют дело не с константами (например, числом π («пи»)), а с текущими значениями переменных (например, текущий цвет шрифта). Достаточно часто присваивание используется сразу при объявлении переменной:

// Объявляем целочисленную переменную х и присваиваем ей значение 5

int x = 5;

При первом присваивании значения переменной говорят об инициализации переменной. В случае одновременного объявления переменной и присваивания ей значения, происходит явная инициализация. Если при объявлении переменной ей не присвоено значение, то она получает значение по умолчанию (неявная инициализация). Например, для переменной логического типа это значение ложь (false).

Теперь рассмотрим другие конструкции языка, позволяющие управлять потоком выполнения программы: условный оператор (if) и операторы цикла (for, while).

Условный оператор позволяет в зависимости от условия выполнить ту или иную последовательность команд. Синтаксис оператора if следующий:

*if**(<условие>) {*

*// Блок кода, который будет выполнен только в случае выполнения условия }*

После ключевого слова if в скобках записывается условие, которое проверяется при выполнении программы, и, в зависимости от его результата, выполняется или не выполняется участок кода, помещённый внутрь фигурных скобок. Условное выражение всегда должно возвращать true или false, т.е. значение типа bool. Это может быть переменная логического типа (bool), выражение с использованием переменных и/или констант (например, x > y) или функция, возвращающая bool.

Оператор if расширяется за счёт использования комбинаций с else и if, который позволяет описать условное или безусловное выполнение другого блока кода. Например:

// Переменная указывающая скорость автомобиля

int carSpeed = 100;

// Если скорость больше 120

if (x > 120) {

Console.WriteLine("Скорость больше 120. Лишение прав!");

}

// Если скорость больше 100

else if (x > 100) {

Console.WriteLine("Скорость больше 100. Штраф!");

}

// Во всех других случаях (меньше или равна 100)

else {

Console.WriteLine("Нет нарушения");

}

В некоторых случаях удобно записать ветвление с помощью оператора switch. Он хорошо подходит, когда в зависимости от значений одной переменной необходимо выполнить разные инструкции. В примере ниже переменная typeFullName (описывает путь к типу в C#) получает значение в зависимости от значений переменной typeAlias (описывает сокращённое имя типа в C#):

// Объявляем строковую переменную и присваиваем ей значение

string typeAlias = "int";

// Объявляем строковую переменную со значением по умолчанию

string typeFullName;

switch (typeAlias) {

case ("int"): typeFullName = "System.Int32"; break;

case ("double"): typeFullName = "System.Double"; break;

case ("string"): typeFullName = "System.String"; break;

default: typeFullName = "Неизвестный алиас"; break;

}

// Вывод результата

Console.WriteLine(typeAlias);

Console.WriteLine(typeFullName);

Синтаксис оператора switch следующий:

*switch (<переменная, значения которой рассматриваются>) {*

*case (<вариант значения переменной>):*

*<соответствующий блок инструкций> break;*

*case (<другой вариант значения переменной>):*

*<соответствующий блок инструкций > break;*

*default:*

*<блок инструкций, выполняемый, если значение переменной не соответствует ни одному варианту, представленному выше> break;*

*}*

Наконец, последняя операция, без которой нельзя описать компьютерную программу, это цикл, т.е. повторение блока инструкций несколько раз. В языке C#, впрочем, как и во многих других языках, для этой цели предусмотрено несколько ключевых слов: for, while, do while.

Цикл for описывается с помощью следующего синтаксиса:

*for (<Раздел инициализатора>; <Раздел условия>; <Раздел итератора>) { <тело цикла> }*

Обычно в разделе инициализатора объявляют и задают начальное значение переменной, которая будет счётчиком повторений (счётчик итераций). В разделе условия задают условие завершения, например, сравнивают счётчик повторений с другим значением. В разделе итератора изменяют значение счётчика.

Цикл for хорошо подходит для перебора элементов массива (набора однотипных значений). Например:

// Инициализация массива строк

string[] companies = { "microsoft", "apple", "samsung" };

for (int i = 0; i < companies.Length; i = i + 1) {

// Формирование строки вывода в формате "№<номер строки> : <наименование компании>"

string outputValue = "№" + i + " : " + companies[i];

Console.WriteLine(outputValue); }

В данном примере перебираются значения массива строк (companies) и в теле цикла производится их форматированный вывод. Счётчиком является переменная i, которая инициализируется значением 0 (индекс первого элемента массива). Условием выполнения является не превышение счётчиком размера массива (companies.Length). В разделе итератора значение счётчика увеличивается на единицу (i = i + 1).

Для увеличения (инкрементирования) значения счётчика можно использовать оператор инкремента: i++.

Более удобный и гибкий подход для форматирования текста заключается в использовании вспомогательного метода String.Format, позволяющего описать шаблон для формирования текста. Шаблон задаётся строкой с «подстановками» в виде «{0}». Следующие далее параметры позволяют через запятую указать значения, которые будут подставлены вместо соответствующих подстановок. Например:

string outputValue = String.Format("№{0} : {1}", i, companies[i]);

Вместо {0} будет подставлено значение переменной i (счётчик), а вместо {1} — элемент массива companies с индексом i. Метод String.Format предоставляет богатый функционал по форматированию текста: позволяет задать ширину строки для подстановки, определить выравнивание подстановки, использовать специфические варианты форматирования для различных типов и др. Метод Console.WriteLine совместим с интерфейсом String.Format и позволяет форматировать вывод без указания метода String.Format:

Console.WriteLine("№{0} : {1}", i, companies[i]);

Помимо конструкции for для организации цикла могут быть использованы и другие. Например, перебор строк, представляющих названия компаний, можно написать с применением конструкции while:

// Синтаксис:

// while <условие завершения> { <тело цикла> }

int i = 0;

while (i < companies.Length) {

Console.WriteLine("№{0} : {1}", i, companies[i]);

i++; }

Или do while:

// Синтаксис:

// do { <тело цикла> } while (<условие завершения >);

int i = 0;

do {

Console.WriteLine("№{0} : {1}", i, companies[i]);

i++; } while (i < companies.Length);

Запись с использованием do while отличается от for и while тем, что тело цикла будет запущено хотя бы 1 раз, даже если условие не выполнено. Такое поведение обусловлено тем, что условие завершения проверяется после выполнения тела цикла. Поэтому do while называют циклом с постусловием. Иногда такая логика работы программы может быть полезна, а иногда – стать причиной ошибки. Например, если исходный массив companies будет пуст, то при первом выполнении тела цикла при использовании do while, произойдёт ошибка. Проблема возникнет при обращении к нулевому элементу массива, которого массив не содержит.

Завершим раздел таблицей соответствия между структурой управления, необходимой для описания алгоритма, и механизмом языка C#:

|  |  |
| --- | --- |
| Таблица 2.1. | |
| **Структура управления** | **Механизм C#** |
| Последовательность | Разделение инструкций точкой с запятой |
| Ветвление | Ключевые слова if, switch |
| Цикл | Ключевые слова for, while, do while |

Механизмы C# для формирования ветвления и цикла не ограничиваются представленными в таблице. Например, ветвление может быть описано с помощью тернарного оператора (<условие> ? <действие при выполнении условия> : <действие при не выполнении условия>), а повторное выполнение блока кода с помощью ключевого слова foreach или рекурсии.

## Контрольные вопросы и задания

1. Дайте определение понятию «переменная».
2. Перечислите основные типы данных. Что определяет тип переменной?
3. Какие необходимы структуры управления для описания любого алгоритма?
4. Приведите синтаксис и семантику условного оператора.
5. Объясните разницу в использовании оператора цикла «do while» от операторов «for» и «while».
6. Перечислите методы для работы с консольным окном для выполнения следующих операций: вывод текста, вывод текста с переходом на следующую строку, ожидание ввода пользователем одного символа, ожидание ввода пользователем строки.
7. Реализуйте программу, которая запрашивает имя пользователя и приветствует его.
8. Реализуйте программу «Флаг», выводящую на экран флаг государства, первая буква которого соответствует первой букве вашей фамилии. Например, студент Русаков должен вывести флаг России.

# Основы объектно-ориентированной разработки на языке C#

## Понятие класса

Основой любого объектно-ориентированного (ООП) языка является понятие класса. Класс — это абстрактный тип данных, включающий в себя набор данных и методов для работы с ними. Например, классом является Личность (Person), содержащая поля: Фамилия, Имя, Отчество и метод для получения полного имени в формате Фамилия И.О.

Если для полей класса заданы конкретные значения — это объект. Иванов Иван Иванович — это объект (или экземпляр) класса Личность. Класс является шаблоном, которому должны соответствовать все объекты, принадлежащие к данному классу. Классы и объекты позволяют объединять логически связанные понятия. Если раньше (до ООП) для полей, связанных с ФИО, необходимо было заводить три переменных, то с появлением ООП для этих целей предполагается использование одного экземпляра класса с тремя полями. Это повышает читаемость и понимание исходного кода.

Для описания класса используется ключевое слово class, после которого следует название класса. Далее в фигурных скобках описываются поля и методы класса. Поля определяются с помощью области видимости (public, protected или private, о них будет рассказано ниже), типа и названия. Для методов задаётся область видимости, тип возвращаемого значения, название и список аргументов в круглых скобках, после которых описывается тело метода. Например:

// Класс личность

public class Person {

public string Name; // Имя

public string Surname; // Фамилия

public string Patronymic; // Отчество

// Метод получения строки «Фамилия И.О.»

public string GetFIO() {

string fio = String.Format("{0} {1}.{2}.", Surname, Name[0], Patronymic[0]);

return fio;

} }

После объявления класса можно объявлять переменные соответствующего типа и создавать новые экземпляры с использованием оператора new. Для обращения к полям и методам класса следует после переменной, указывающей на экземпляр класса, поставить символ «.» (точка) и указать требуемый элемент.

Следующий код выведет на экран «Иванов И.И.»:

Person p1 = new Person();

p1.Name = "Иван";

p1.Surname = "Иванов";

p1.Patronymic = "Иванович";

Console.WriteLine(p1.GetFIO());

Console.ReadKey();

Для задания начальных значений полей класса служит конструктор — метод, вызываемый при создании экземпляра класса. Конструктор задаётся в виде метода без возвращаемого значения. Имя конструктора должно совпадать с именем класса. Каждый класс имеет конструктор по умолчанию, не имеющий аргументов. Этот конструктор может не отображаться в исходном коде (неявный конструктор), но стоит помнить о том, что именно он вызывается при создании экземпляра класса. Помимо конструктора по умолчанию существуют конструкторы, принимающие различные параметры. Для примера, опишем конструктор класса Личность, который инициализирует фамилию, имя и отчество на основании переданных параметров:

public Person(string name, string surname, string patronymic) {

this.Name = name;

this.Surname = surname;

this.Patronymic = patronymic;

}

Если определён хотя бы один конструктор с параметрами, для использования конструктора без параметров необходимо его явное описание:

// Конструктор по умолчанию

public Person() { }

Ключевое слово this позволяет сослаться на экземпляр класса внутри методов этого класса. Выше оно используется, чтобы подчеркнуть различие аргументов конструктора и внутренних полей класса. Объявив вспомогательный конструктор, его можно использовать в программе для передачи параметров при создании экземпляра класса:

Person p1 = new Person("Иван","Иванов","Иванович");

Применение объектно-ориентированного подхода при разработке компьютерных программ даёт множество преимуществ. Обычно с ООП связывают инкапсуляцию, наследование, полиморфизм и абстракцию.

## Инкапсуляция

Инкапсуляция позволяет скрывать от пользователя класса (другого программиста) внутреннюю реализацию. Сделать это можно, применив специальные модификаторы доступа для полей или методов класса. В примере выше перед методом GetFIO использовано ключевое слово public, которое разрешает вызов метода всем, у кого имеется доступ к экземпляру класса Личность. Но если использовать ключевое слово private, то вызвать метод не удастся, так как он инкапсулирован и доступен только внутри класса, т.е. к нему обращаются только методы класса, в котором он описан. Аналогичная ситуация и с объявлением полей класса: помеченные ключевым словом private будут недоступны «снаружи».

Недоступные для внешнего использования поля и методы реализуют внутреннее поведение класса и инкапсулируют (скрывают) логику работы от пользователей класса. Среда Visual Studio отслеживает обращение к «недоступным» элементам класса и сообщает об ошибке.

## Наследование

Другим преимуществом применения классов ООП является наследование, которое позволяет повысить возможность повторного использования кода. С помощью наследования новый класс создаётся не «с нуля», а на основе другого, базового класса. При этом класс-наследник получает часть функциональности базового класса. Например, если описать класс Студент, сделав его наследником класса Личность, то можно использовать функционал, связанный с ФИО и реализованный в базовом классе Личность. Для определения базового класса следует указать его название после имени класса-наследника через двоеточие:

// Класс наследник

public class Student: Person {

// Конструктор

public Student(string name, string surname, string patronymic)

// Вызываем конструктор базового класса

:base(name,surname,patronymic) { }

// Имя группы

public string GroupName;

// Получить ФИО с группой

public string GetFIOWithGroup() {

// Использование метода GetFIO() базового класса

string nameWithGroup = String.Format("{0} {1}", this.GetFIO(), this.GroupName);

return nameWithGroup;

} }

Для ограничения наследования методов и полей используются модификаторы доступа. Поле или метод, помеченный как private, не будет наследоваться, а модификатор protected указывает, что метод или атрибут наследуется, но доступен только внутри базового класса и класса-наследника. В примерах выше использован модификатор public, разрешающий доступ «снаружи» и указывающий на наследование поля или метода. Если не задать модификатор доступа, то по умолчанию он считается как private.

Создавая целые иерархии классов, которые будут мигрировать из проекта в проект, можно существенно сократить трудоёмкость решения типовых задач.

## Полиморфизм

Полиморфизм — это способность объектов, имеющих одинаковый базовый класс, по-разному реализовывать его методы. Рассмотрим полиморфизм на примере — объявим в классе Личность метод WriteName, который будет выводить ФИО личности, и пометим его как virtual. Ключевое слово virtual говорит наследникам класса о том, что данный метод поддерживает переопределение, т.е., вероятно, будет иметь специфическую для наследника реализацию. Для переопределения базовой реализации метода необходимо пометить метод как override.

public class Person {

// Виртуальный метод

virtual public string GetName()

{ return this.GetFIO(); }

}

public class Student: Person {

// Переопределение базовой реализации метода

override public string GetName()

{ return this.GetFIOWithGroup(); }

}

Теперь создадим массив объектов базового класса, в который будут входить как личности, так и студенты, и в цикле для всех вызовем метод WriteName. Для объектов класса Студент будет вызвана та реализация, которая указана в классе-наследнике, а не в базовом.

Person p1 = new Person("Иван","Иванов","Иванович");

Student st1 = new Student("Андрей", "Петров", "Иванович");

st1.GroupName = "А-13-05";

Person[] persons = {p1, st1};

foreach (Person person in persons) {

string name = person.GetName();

Console.WriteLine(name);

}

На экран будет выведено:

***Иванов И.И.***

***Петров А.И. А-13-05***

Наследование и полиморфизм применяются для увеличения гибкости и возможности повторного использования программного кода. Например, если необходимо вывести на экран имя объекта, то методу, выполняющему эту операцию, совершенно необязательно знать конкретный класс объекта. Достаточно того, что класс объекта является наследником класса, который реализует метод получения имени — GetName.

Помимо классов для задания общего поведения всех наследников могут быть использованы интерфейсы. В некотором смысле, это подобие базовых классов, но без описания полей и реализации методов. Интерфейс содержит только декларацию (объявление) методов. Если для класса указан интерфейс, то, говорят, класс реализует этот интерфейс.

Наследование и полиморфизм позволяют описывать новые классы и использовать их в реализованных ранее методах за счёт соответствия определённому базовому классу или интерфейсу. Например, у вас возникла необходимость представить объект в виде последовательности битов для сохранения в файл. Для добавления такой функциональности своему классу, следует реализовать интерфейс ISerializable. После чего с экземплярами класса смогут работать методы, осуществляющие запись в файл. При этом реализация взаимодействия с файловой системой инкапсулирована (скрыта). Вам, как пользователю классов, обеспечивающих работу с файловой системой, доступны лишь некоторые методы, помеченные как public и взаимодействующие с классами, реализующими интерфейс ISerializable.

## Абстракция

Наконец, абстракция — это подход при разработке программного обеспечения в целом, который был сформирован независимо от появления ООП. Абстракция позволяет убрать из рассмотрения характеристики объектов реального мира, которые никак не влияют на работу вашей программы. Например, если мы хотим сгруппировать студентов по возрасту, то нам совершенно не важен их вес или рост (достаточно даты рождения). Если же разрабатываемая программа связана с карточкой пациента поликлиники, то в ней должно быть больше данных, и должны присутствовать поля вес и рост. Но и в этом случае есть характеристики, которые следует исключить. Например, название оператора мобильной связи, которым пользуется пациент, или баланс его лицевого счёта. В обоих случаях мы абстрагируемся от реального мира и строим его модель, с которой будет работать наша программа.

Абстракция была всегда связана с программированием, она появилась задолго до ООП, и умение её использовать в большей мере зависит от навыков программиста, нежели от языка программирования, который он использует. Однако с приходом ООП некоторые вещи стало описывать намного легче. Например, базовый класс позволяет вынести общий для всех потомков функционал и собрать его в одном месте, а интерфейс — определить (специфицировать) общее поведение классов, не спускаясь на уровень реализации.

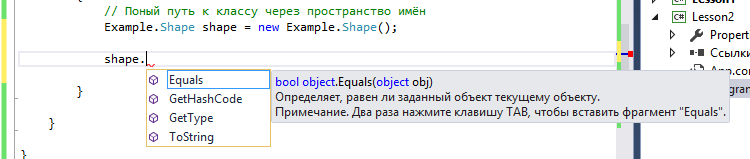
На этом завершим обзор базовых типов, конструкций языка для управления потоком и основ ООП с применением C#. Отдельно отметим, что в предложенном материале даны самые основы языка C#. Более глубоко «окунуться» в мир разработки с использованием C# поможет книга Джеффри Рихтера [5], которая считается классикой для разработчиков на платформе .NET. Перейдём к рассмотрению возможностей среды Microsoft Visual Studio для создания приложений.

## Контрольные вопросы и задания

1. Дайте определение понятию «класс».
2. Приведите пример класса и объекта этого класса.
3. Перечислите возможные области видимости в языке C#.
4. В чём заключается принцип инкапсуляции?
5. Для каких целей применяется наследование классов?
6. Дайте определение понятия «полиморфизм».
7. В чём заключается применение абстракций при разработке программного обеспечения?
8. Реализуйте программу «Флаг» в терминах объектно-ориентированного программирования.

# Основы работы в Visual Studio

Мы уже создали простейшее приложение в среде разработки Visual Studio и знаем, как его запустить. В процессе разработки был задействован ряд сложных технологий. Они кажутся привычными для современного программиста, но на самом деле появились они относительно недавно и за их реализацией скрывается огромная работа. Например, при редактировании исходного кода среда разработки «подсказывает» возможные варианты продолжения инструкции (см. рис. 4.1). Так работает технология автоматического дополнения печати, которая называется IntelliSense. Разработка программы связана с набором большого количества строк исходного кода и IntelliSense облегчает эту работу за счёт дописывания названий функций, переменных и других составляющих исходного кода программы при вводе начальных букв. Другой плюс заключается в том, что при вводе во всплывающем окне отображаются не только подходящие элементы (все, если начальная строка пуста), но и их краткое описание. В случае метода в описание входит информация о параметрах, необходимых для вызова.



**Рис. 4.1. Автодополнение IntelliSense**

Среда Visual Studio поддерживает добавление такого описания и к собственным методам. Этот механизм называется «самодокументирование».

Для создания приложений в среде Visual Studio необходимо уметь работать со структурой решения и транслировать исходный код программы в машинный код, выполняемый компьютером. Познакомимся с этими возможностями.

## Работа со структурой решения

При создании нового приложения Visual Studio одновременно создаёт решение и проект, затем включает проект в решение. На верхнем уровне группировки кода находится решение. Далее идут проекты.

Физически решение представляет собой набор различных файлов в файловой системе. Visual Studio представляет разработчику эти файлы в структурированном и удобном для разработки виде. Структура текущего решения отображается в окне «Обозреватель решений» («Вид → Обозреватель решений»). Обозреватель решений позволяет редактировать структуру, добавляя и удаляя новые проекты, папки и файлы.

Решение состоит из нескольких проектов, которые логически связаны друг с другом. Например, один проект содержит классы, позволяющие рисовать геометрические фигуры на экране, другой — решать геометрические задачи, а третий организует взаимодействие с пользователем и использует классы из первых двух проектов как вспомогательные для решения и графического оформления поставленных пользователем задач.

Для добавления в решение нового проекта следует выбрать в контекстном меню (открывается при нажатии правой кнопки мыши на нужном объекте) верхнего уровня (уровень решения) пункт «Добавить → Создать проект». Кроме создания нового проекта можно добавить существующий проект. После добавления нового проекта он будет отображен в обозревателе решений с характерной иконкой.

Когда решение содержит несколько проектов, необходимо определить какой из проектов будет стартовать при запуске решения (нажатие на кнопку «Пуск»). Сделать это следует из контекстного меню для нужного проекта, выбрав пункт «Назначить запускаемым проектом».

В структуре решения под проектом находится множество элементов. Это могут быть файлы с исходным кодом классов или интерфейсов на различных языках программирования (c#, c++, visual basic и другие), файлы гипертекстовой разметки html, каскадных стилей css, файлы сценариев на языке javascript, файлы изображений, файлы ресурсов для поддержки различных языков и многие другие типы документов.

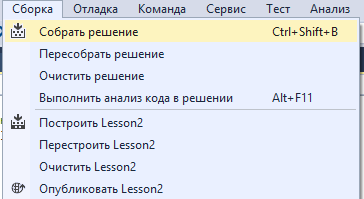
Добавить новый или существующий элемент в проект можно через контекстное меню: «Добавить → Создать элемент/Существующий элемент». Наиболее используемые элементы можно создать, сразу выбрав их тип, например, «Добавить → Класс».

Желательно декомпозировать (разделять) исходный код для облегчения восприятия и редактирования (принцип «разделяй и властвуй»). Удобно иметь в одном файле один класс. Конечно, если классы имеют минимальный набор полей и методов и тесно связаны, они могут быть размещены в одном файле. Также существуют вложенные классы, т.е. классы, объявленные внутри другого класса. В этом случае их описание находится в одном файле.

Когда в проекте множество файлов (например, классы для разных геометрических фигур), работа усложняется большим количеством открытых для редактирования документов. Чтобы быстро уменьшить их количество, следует открыть контекстное меню для заголовка открытого файла и выбрать одну из опций: «Закрыть все документы», «Закрыть все, кроме этого».

## Сборка и запуск решения

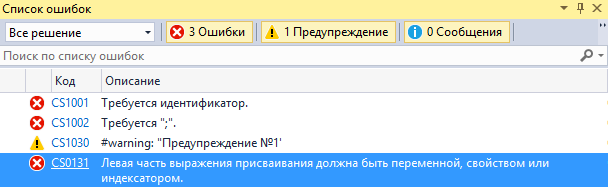
Конечной целью написания исходного кода является его запуск, а для этого код необходимо скомпилировать (транслировать на язык, понятный компьютеру). В результате компиляции или сборки проекта появляется исполняемый файл, который понятен компьютеру и может быть им выполнен. Функции по сборке доступны из соответствующего раздела меню «Сборка → Собрать решение» или «Сборка → Построить проект»:



**Рис. 4.2. Запуск сборки решения**

В процессе разработки периодически следует запускать компиляцию исходного кода для проверки его на наличие ошибок. Наиболее распространены синтаксические ошибки (наличие неправильных конструкций языка программирования), но могут быть и другие. Например, задана ссылка на ресурс, не включённый в состав проекта, или отсутствует доступ на запись для результирующего файла.

Процесс сборки отображается в окне «Вывод», а ошибки, в случае их наличия, — в окне «Список ошибок» (см. рис. 4.3). Если эти окна не отображаются, открыть их можно через меню: «Вид → Список ошибок» и «Вид → Вывод». Все обнаруженные проблемы при сборке решения разделяются на ошибки, предупреждения и сообщения. При наличии хотя бы одной ошибки исполняемый файл не может быть получен и актуальная версия приложения не может быть запущена.



**Рис. 4.3. Окно ошибок и предупреждений**

При нажатии на кнопку «Пуск» по умолчанию осуществляется сборка всех проектов, входящих в решение, и запускается проект, назначенный запускаемым. В случае если в процессе сборки произошли ошибки, и системе не удалось получить новую версию, вам будет предложено запустить последнюю имеющуюся сборку.

Частой проблемой для новичков является положительный ответ на этот вопрос. В этом случае, даже если в текущей версии исходного кода есть ошибки, будет запущена предыдущая версия приложения. Это путает начинающего разработчика, так как исходный код и запускаемое приложение не будут синхронизированы. Рекомендуется ответить на данный вопрос отказом и установить «галку» для опции «Больше не выводить это окно». Чтобы изменить принятое решение, необходимо зайти в раздел «Отладка → Параметры», пункт «Проекты и Решения → Сборка и запуск», опция «При выполнении, когда возникают ошибки сборки или развертывания».

## Контрольные вопросы и задания

1. Как работает технология IntelliSense?
2. Какое окно предоставляет доступ к просмотру структуры решения Visual Studio?
3. Перечислите элементы в порядке вложенности, начиная от наиболее «крупного»: метод, проект, решение, класс.
4. Перечислите типы возможных проблем при сборке проекта. Наличие проблемы какого типа не позволит запустить проект?
5. Разделите реализацию программы «Флаг» так, чтобы каждый класс находился в отдельном файле.

# Разработка консольного меню

## Проектирование

В качестве практического примера применения освоенного материала реализуем набор вспомогательных классов для создания консольного меню приложения. Поставим цель создать «каркас», который подходил бы для любого приложения, работающего в консольном режиме, т.е. абстрагируемся от конкретной задачи. Пусть меню будет поддерживать иерархическую вложенность разделов, чтобы не «пугать» пользователя отображением всех возможностей сразу, а предоставлять ему возможность постепенно углубляться и находить нужный функционал. При навигации по разделам и после выполнения действий отдельных элементов, экран должен очищаться так, чтобы пользователю отображался минимально необходимый набор информации.

Результат должен выглядеть следующим образом:

1 - Пункт меню 1

2 - Пункт меню 2

3 - Пункт меню 3

0 - Выход (на уровень вверх)

Для каждого из пунктов меню должно быть назначено действие для запуска, либо описаны вложенные разделы.

Определим классы и интерфейсы, которые нам потребуются. Во-первых, это класс, представляющий собой всё меню, — Menu. Во-вторых, нам потребуется иерархия классов для элементов меню с различной функциональностью. Пусть все они реализуют интерфейс IMenuItem, с которым будет работать класс Menu. Хорошим тоном является наличие на верхнем уровне любой иерархии классов одного или нескольких интерфейсов, так как они являются более общим случаем класса и не содержат никакой реализации. Именно на уровне интерфейсов должен проходить этап проектирования программного обеспечения. В нашем примере мы могли бы выделить интерфейс для класса Menu, но не будем этого делать в силу небольшого размера проекта, чтобы не затруднять его восприятие.

Интерфейс IMenuItem будет требовать от реализующих его классов следующую функциональность:

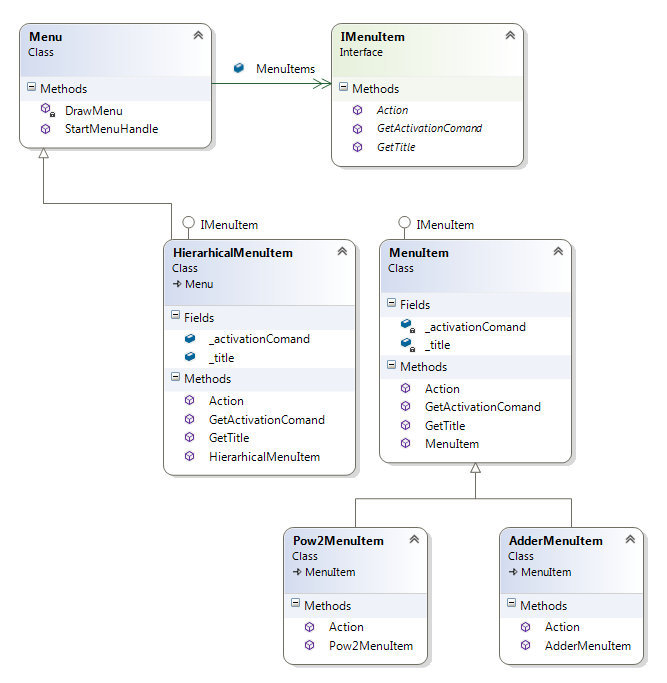
1. Получение заголовка для вывода в меню (метод GetTitle).
2. Получение команды, по которой активируется действие (метод GetActivationComand).
3. Метод для запуска действия, соответствующего элементу меню (метод Action).

Первые два метода возвращают строковый тип данных, а у второго возвращаемое значение отсутствует. Меню будет содержать в себе коллекцию экземпляров (список MenuItems), классы которых реализуют интерфейс IMenuItem, и использовать их для вывода меню пользователю (метод DrawMenu). Помимо этого, меню отвечает за обработку пользовательского ввода и запуск действия вызываемых элементов меню. Переход в этот режим «диалога» с пользователем будет осуществляться с помощью вызова метода StartMenuHandle.

Для проверки работоспособности меню потребуется реализовать хотя бы один класс с интерфейсом IMenuItem. Таким классом будет MenuItem. Он будет предоставлять возможность задания заголовка, содержать команды для активации действия, а также выполнять простейшее действие — выводить свой заголовок в консоль. Впоследствии этот класс может выступать в роли базового для других классов, которым потребуется быть элементами меню. В нашем примере такими классами-наследниками будут: Pow2MenuItem (возведение числа в квадрат) и AdderMenuItem (сложение двух чисел).

Для поддержки вложенности меню реализуем класс HierarhicalMenuItem, логика работы которого соответствует логике самого меню — вывод меню и обработка пользовательской команды. Поэтому класс HierarhicalMenuItem помимо реализации интерфейса IMenuItem будет являться наследником класса Menu.

Более наглядно классы, интерфейсы и их взаимосвязи позволяет представить диаграмма классов языка UML (Unified Modelling Language):



**Рис. 5.1. Диаграмма классов**

## Разработка

После предварительного обсуждения и выполненного этапа проектирования с помощью диаграммы классов можно переходить к реализации. Исходный код интерфейса элементов меню выглядит следующим образом:

public interface IMenuItem {

// Метод, возвращающий заголовок меню

string GetTitle();

// Метод, возвращающий команду, активирующую действие

string GetActivationComand();

// Запуск действия соответствующего элементу

void Action(); }

Класс Menu ничего не знает о конкретных классах, реализующих интерфейс IMenuItem, и работает только с IMenuItem:

public class Menu {

// Список элементов меню

public List<IMenuItem> MenuItems = new List<IMenuItem>();

// Метод вывода элементов меню и пункта выхода

private void DrawMenu() {

foreach (IMenuItem menuItem in this.MenuItems) {

Console.WriteLine("{0} - {1}", menuItem.GetActivationComand(), menuItem.GetTitle());

}

Console.WriteLine("0 - Назад");

} }

Метод для перехода в режим ожидания пользовательского ввода и запуска соответствующего пункта меню выглядит следующим образом:

public void StartMenuHandle() {

string userInput = null;

this.DrawMenu();

// После отрисовки меню курсор находится на строке, где пользователь будет вводить данные. Запоминаем эту позицию, чтобы потом очистить консоль и вернуться в исходное состояние

int drawMenuLastLineIndex = Console.CursorTop;

while (userInput != "0") {

// Просим пользователя ввести значение и запоминаем его

userInput = Console.ReadLine();

// Поиск элемента меню, для которого подходит команда, введённая пользователем

foreach (IMenuItem menuItem in this.MenuItems) {

if (menuItem.GetActivationComand() == userInput) {

Console.WriteLine("Пользователь выбрал пункт: " + menuItem.GetTitle());

menuItem.Action();

} }

// Позиция курсора после действий дочерних элементов

int actionLastLineIndex = Console.CursorTop + 1;

// Очистка возможных строк, добавленных при выполнении действия

ConsoleHelper.ClearLines(drawMenuLastLineIndex, actionLastLineIndex);

} }

Чтобы иметь возможность создавать элементы меню, создадим базовый класс MenuItem, реализующий интерфейс IMenuItem:

public class MenuItem : IMenuItem {

// Поле для хранения команды, активирующей действие

private string \_activationComand;

// Поле для хранения названия пункта меню

private string \_title;

// Конструктор для создания элемента меню

public MenuItem(string title, string activationComand) {

this.\_title = title;

this.\_activationComand = activationComand; }

// Реализация интерфейса IMenuItem

public string GetActivationComand()

{ return this.\_activationComand; }

public string GetTitle()

{ return this.\_title; }

// Виртуальный метод. Может быть переопределён в наследниках

virtual public void Action() {

Console.WriteLine("Действие для '{0} - {1}'", this.\_title, this.\_activationComand);

Console.WriteLine("Нажмите любую клавишу для продолжения...");

Console.ReadKey();} }

Теперь приведём исходный код класса HierarhicalMenuItem, который позволяет организовать вложенность элементов меню. Так как поведение класса HierarhicalMenuItem схоже с поведением класса Menu, одним из вариантов реализации HierarhicalMenuItem является наследование функциональности класса Menu. Другой вариант поддержки вложенности элементов меню — это поддержка интерфейса IMenuItem в классе Menu, так, чтобы само меню смогло выступать в роли одного из элементов. Для большей наглядности остановимся на первом варианте:

public class HierarhicalMenuItem : Menu, IMenuItem {

// Реализация конструктора и методов GetActivationComand и GetTitle интерфейса IMenuItem по аналогии с классом MenuItem

…

// Переопределение метода-действия

public void Action() {

// Вызов метода класса-родителя (меню) для вывода меню и перехода в режим обработки пользовательских команд

this.StartMenuHandle();} }

В этой реализации видна вся мощь объектно-ориентированного подхода к разработке программного обеспечения. Ключевое место в исходном коде занимает объявление класса, его базового класса и поддерживаемого интерфейса. В классе HierarhicalMenuItem нет ни строчки кода, связанной с реализацией меню, вся необходимая функциональность взята из базового класса. Можно сказать, что класс HierarhicalMenuItem адаптирует класс Menu так, чтобы он реализовывал интерфейс IMenuItem.

## Пример использования

В качестве примера применения разработанных классов рассмотрим создание меню, предлагающего пользователю выполнить простые арифметические операции обработки данных.

Каждая из операций представляет собой элемент меню и должна реализовывать интерфейс IMenuItem. Наиболее простой способ обеспечить этот функционал — унаследовать поведение класса MenuItem, переопределив в нём метод Action. Например, описание класса Pow2MenuItem, который позволяет пользователю возвести число в квадрат, выглядит следующим образом:

public class Pow2MenuItem : MenuItem {

// Конструктор, который вызывает базовый конструктор MenuItem

public Pow2MenuItem(string title, string activationComand)

: base(title, activationComand) { }

// Переопределяем действие

public override void Action() {

Console.Write("Введите число ");

string userInput = Console.ReadLine();

int number = 0;

// Если строку не удалось преобразовать в число

if (!int.TryParse(userInput, out number)) {

Console.WriteLine("Введённое значение '{0}' не является числом", userInput);

return; }

Console.WriteLine("Квадрат числа '{0}' равен '{1}'", number, number \* number);

Console.WriteLine("Нажмите любую клавишу для продолжения...");

Console.ReadKey(); } }

Статические методы TryParse и Parse класса int пытаются преобразовать строку в число. В случае ошибки Parse выдаёт исключительную ситуацию, а TryParse возвращает false. Предпочтительней использовать метод TryParse.

Метод Action класса AdderMenuItem запрашивает у пользователя 2 числа и складывает их:

public override void Action() {

Console.Write("Введите первое число: ");

string userInput1 = Console.ReadLine();

int number1 = 0;

if (!int.TryParse(userInput1, out number1)) {

Console.WriteLine("Введённое значение '{0}' не является типом Int", userInput1);

return; }

Console.Write("Введите второе число: ");

string userInput2 = Console.ReadLine();

int number2 = 0;

if (!int.TryParse(userInput2, out number2)) {

Console.WriteLine("Введённое значение '{0}' не является типом Int", userInput2);

return; }

Console.WriteLine("Результат: '{0}'", number1 + number2);

Console.WriteLine("Нажмите любую клавишу для продолжения...");

Console.ReadKey();}

После того как реализованы отдельные элементы меню, необходимо создать экземпляр класса меню и наполнить его элементами. Предположим, что на верхнем уровне пользователю предлагается выбрать между арифметическими операциями и операциями со строками. Для этого создадим два соответствующих раздела. Раздел «Арифметические операции» наполним экземплярами только что реализованных классов. Итоговый листинг программы выглядит следующим образом:

static void Main(string[] args)

{

Menu menu = new Menu();

// Наполняем меню элементами

HierarhicalMenuItem hierarhicalMenuItemArifmetic = new HierarhicalMenuItem("Арифметические операции", "1");

menu.MenuItems.Add(hierarhicalMenuItemArifmetic);

HierarhicalMenuItem hierarhicalMenuItemString = new HierarhicalMenuItem("Операции над строками", "2");

menu.MenuItems.Add(hierarhicalMenuItemString);

// Наполняем иерархический элемент меню

hierarhicalMenuItemArifmetic.MenuItems.Add(new Pow2MenuItem("Возведение в квадрат", "1"));

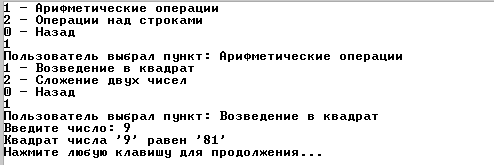
hierarhicalMenuItemArifmetic.MenuItems.Add(new AdderMenuItem("Сложение двух чисел", "2"));

// Переход в режим обработки пользовательских запросов

menu.StartMenuHandle();

}

Пример работы представлен на рисунке:



**Рис. 5.2. Пример работы меню**

## Контрольные вопросы и задания

1. Какой класс отвечает за поддержку вложенных элементов меню?
2. Какие методы декларирует интерфейс IMenuItem?
3. Почему класс MenuItem определяет метод Action как виртуальный?
4. Реализуйте статический метод ClearLines класса ConsoleHelper, очищающий заданный диапазон строк и устанавливающий позицию курсора в начальную позицию.
5. «Соберите» описанную реализацию в собственное приложение и расширьте его дополнительными арифметическими и строковыми операциями.

# Заключение

Разработанное решение достаточно универсально, реализовано с применением ООП подхода в отрыве от контекста конкретной задачи. Разработка велась в среде MS Visual Studio на языке программирования C#. Созданный «каркас» меню может быть использован для любой задачи, предлагающей пользователю интерфейс взаимодействия через консольное окно.

# Библиографический список

1. Загрузка среды разработки MS Visual Studio: <https://www.visualstudio.com/> .
2. Создание учётной записи Microsoft: <https://login.live.com/RU> .
3. **Рамбо, Д.** UML 2.0. Объектно-ориентированное моделирование и разработка : пер. с англ. / Д. Рамбо, М. Блаха . – 2-е изд . – СПб. : Питер, 2007 . – 544 с. – (Б-ка программиста) . - ISBN 5-469-00814-2 .
4. Microsoft Developer Network: <https://msdn.microsoft.com/> .
5. **Рихтер Дж.** CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е изд. – СПб.: Питер, 2012. – 928 с.: ил. ISBN 978-5-459-00297-3 .
6. **Троелсен, Э.** C# и платформа .NET : пер. с англ. / Э. Троелсен . – СПб. : Питер, 2007 . – 796 с. – (Б-ка программиста) . – ISBN 5-318-00750-3 .
7. **Макаров, А. В.** Common Intermediate Language и системное программирование в Microsoft. NET : учебное пособие для вузов по направлению "Информатика и вычислительная техника" / А. В. Макаров, С. Ю. Скоробогатов, А. М. Чеповский . – М. : Интернет-Ун-т информ. технологий : БИНОМ. Лаборатория знаний, 2010 . – 328 с. – (Основы информатики и математики) . - ISBN 5-9556-0055-8 .