



**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ**
**ФГАОУ ВО «Севастопольский государственный
университет»**
**Институт радиоэлектроники и информационной
безопасности**

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
практикум по дисциплине
**«Визуальное проектирование программ в сре-
де Microsoft Visual Studio»**
для студентов дневной и заочной форм обучения
направлений подготовки 11.03.04 «Электроника и нано-
электроника»

Часть 1

**Севастополь
2019 г.**

УДК 004.4'236

Учебно-методическое пособие «Практикум по дисциплине «Визуальное проектирование программ в среде Microsoft Visual Studio» Часть 1 для студентов дневной и заочной форм обучения по направлению подготовки 11.03.04 «Электроника и нанoeлектроника» / СевГУ; Сост. к.т.н. Д.Г. Мурзин, Н.Г. Вильсон, А.А. Азаров – Севастополь: Изд-во СевГУ, 2019. – 131 с.

Целью учебно-методического пособия является оказание помощи студентам очной формы обучения в подготовке и выполнении практикума по дисциплине «Визуальное проектирование программ в среде Microsoft Visual Studio».

Рассмотрены и утверждены на заседании кафедры электронной техники (протокол № 5 от 23 мая 2019 г.)

Рецензенты: к.т.н., доцент кафедры электронной техники Шевченко Н.В.

Ответственный за выпуск: заведующий кафедрой «Электронная техника», к.т.н., доцент Михайлюк Ю.П.

СОДЕРЖАНИЕ

| | |
|--|-----|
| Требования к оформлению отчета к практической работе | 4 |
| Введение в объектно-ориентированое программирование... | 5 |
| Практическая работа № 1 | 14 |
| Практическая работа № 2 | 32 |
| Практическая работа № 3 | 68 |
| Практическая работа № 4 | 106 |

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА К ПРАКТИЧЕСКОЙ РАБОТЕ

Отчет о выполнении практической работы оформляется на стандартных листах формата А4. Расположение листов – вертикальное. Поля: левое 2 см, верхнее 2 см, нижнее 1 см, правое 1 см. Отчет следует оформлять в редакторе Microsoft Word.

Отчет должен включать в себя титульный лист с указанием номера и темы практической работы; цель работы и индивидуальное задание; блок-схему алгоритма программы с комментариями; анализ работы алгоритма и выводы (полученные результаты).

Блок-схема выполняется средствами редактора MS Visio и должна отражать весь алгоритм работы программы с комментариями. Если программа состоит из нескольких модулей, то для каждого модуля составляется отдельная блок-схема. В случае если в алгоритме определены объекты, то должна быть представлена структура каждого объекта.

При оформлении блок-схемы алгоритма программы следует придерживаться следующих правил:

- 1) элементы блок-схемы изображаются согласно ГОСТ;
- 2) линии изображаются со стрелками, если они (или какая-либо их часть) направлены вверх или влево;
- 3) пересечения линий не допускаются. В случае, когда возникает ситуация с пересечением линий, следует воспользоваться элементами «узел разрыва линий»;
- 4) линии должны соединяться с фигурами только сверху или снизу. Исключение составляет блок «Решение». Для него разрешается ответвление линий в любую сторону;
- 5) все надписи выполняются шрифтом одного размера; если надписи выполняются «от руки», то используют чертежный шрифт подходящего размера.

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1. Технология программирования в историческом аспекте

Технология программирования – совокупность методов, приемов и средств, обеспечивающих сокращение стоимости и повышения качества разработки программных систем.

«Стихийное» программирование охватывает период от момента появления первых вычислительных машин до середины 60-х годов XX в. В этот период отсутствовали сформулированные технологии, и программирование было искусством. Программы писались на машинном языке и работали с напрямую данными (рис.1). Появление ассемблеров позволило вместо двоичных или шестнадцатеричных кодов использовать символические имена данных и мнемоники кодов операций.



Рис. 1 – Структура первых программ

Революционным было появление в языках средств, позволяющих оперировать подпрограммами. Типичная программа с подпрограммами состояла из основной программы, области глобальных данных и набора подпрограмм, выполняющих обработку всех данных или их части (рис. 2). Проблемой подобной архитектуры являлась вероятность искажения части глобальных данных какой-либо подпрограммой. Для сокращения таких ошибок в подпрограммах начали размещать локальные данные (рис. 3).

Возникновение **структурного программирования** (60-70-е XXв.) связано с именем известного голландского ученого Э. Дейкстры, который в 60-х годах прошлого века сформулировал основные ее положения.

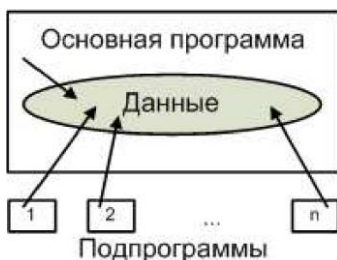


Рис. 2 – Архитектура программы с глобальной областью данных



Рис. 3 – Архитектура программы, использующей подпрограммы с локальными данными

В основе структурного подхода лежит процедурная декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм.

Структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их

описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации. На рис. 4 представлено изображение указанных алгоритмических конструкций в виде блок-схем. Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования, таких как PL/1, ALGOL-68, Pascal, C.

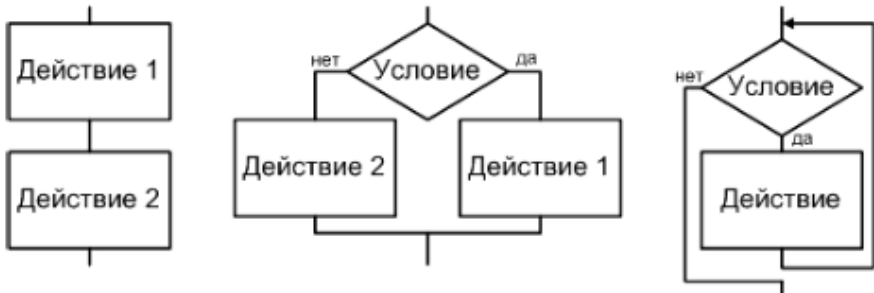


Рис. 4 – Блок-схемы базисных алгоритмических конструкций

Технология модульного программирования, оформившаяся в начале 70-х годов XX, предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (рис. 5), называемые библиотеками подпрограмм. С появлением модульного программирования связаны возможности коллективной разработки программ и повторного использования созданного ранее кода.

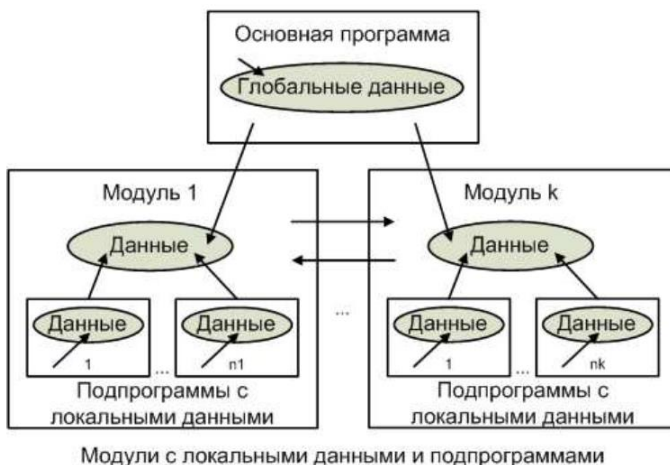


Рис. 5 – Архитектура программы, состоящей из модулей

Эту технологию поддерживают современные версии языков Pascal, C (C++), Ада и Modula.

Объектно-ориентированное программирование (с середины 80-х до конца 90-х гг. XX в.) определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов (Рис. 6), каждый из которых является экземпляром определённого типа (класса), а классы образуют иерархию с наследованием свойств.

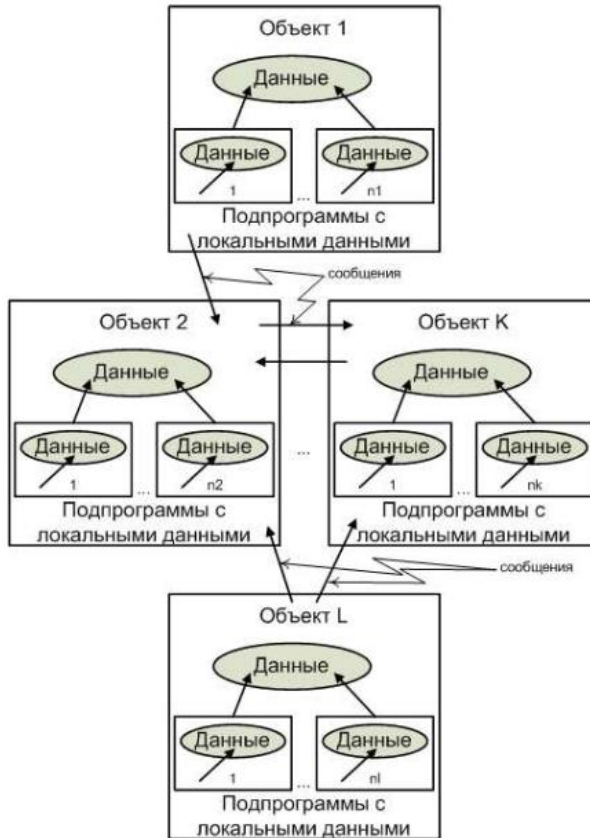


Рис. 6 – Архитектура программы при объектно-ориентированном программировании

Взаимодействие программных объектов в такой системе осуществляется путём передачи сообщений. Объектная структура программы впервые была использована в языке имитационного моделирования Simula (60-х гг. XX в), а затем в языке моделирования – Smalltalk (70-е гг. XX в.), а затем реализована в универсальных языках программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного программирования является «более естественная» декомпозиция программного обеспечения. Это приводит к лучшей локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Объектный подход предлагает способы организации программ, основанные на механизмах абстракции, инкапсуляции, наследовании, полиморфизма, что позволяет конструировать сложные объекты из сравнительно простых, увеличивая показатель повторного использования кодов.

Компонентное программирование (с середины 90-х гг. XX в. до нашего времени) – предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов, объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде и использовать в любом языке программирования, поддерживающем соответствующую технологию.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов) – OLE-automation, ActiveX, и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов).

На современном этапе развития технологии программирования широко используются автоматизированные технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/ System Engineering). Существуют CASE-технологии,

поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

2. Классификация языков программирования

Языки программирования классифицируются (рис. 7): по степени их зависимости от вычислительной машины; по ориентации на сферу применения; по специфике организационной структуры языковых конструкций и т.п.

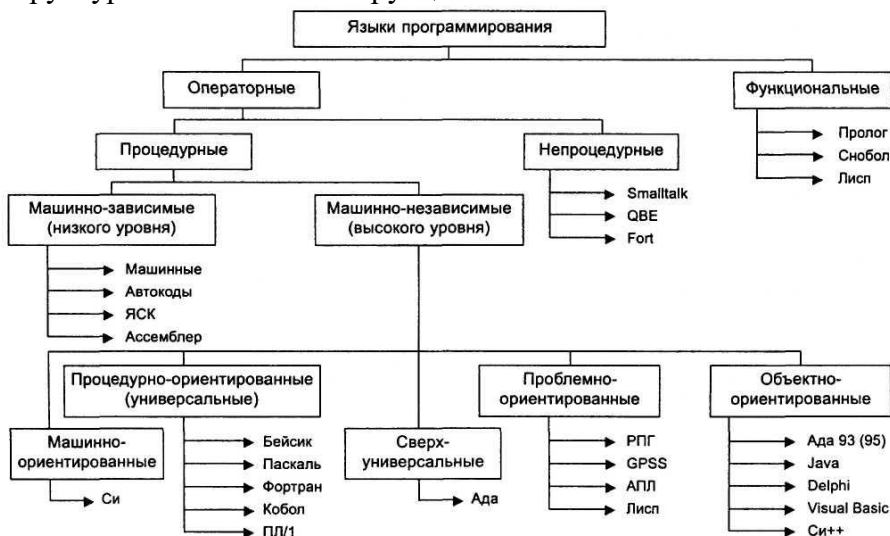


Рис. 7 – Классификация языков программирования

В настоящее время насчитывается более 2000 языков программирования, большинство из которых, возникло исходя из конкретных требований некоторой предметной области. На развитие языков программирования значительное влияние оказали достижения теории вычислений, которые привели к формальному пониманию семантики операторов, модулей, абстрактных типов данных и процедур.

Все языки программирования можно объединить в 4 группы по признаку поддерживаемых ими абстракций: математические, алгоритмические, ориентированные на данные, объектно-ориентированные. На рис. 8 показана генеалогия наиболее распространенных языков программирования и развитие основных концепций программирования.

3. История развития C#

Microsoft Visual C# представляет собой эффективный и простой язык, предназначенный для разработчиков, создающих сборки приложений в среде Microsoft .NET Framework.

Visual C# унаследовал множество лучших свойств от C++ и Microsoft Visual Basic, но при этом его разработчики избавились от различных несоответствий и анахронизмов, в результате чего появился более понятный и логичный язык.

Версия C# 1.0 дебютировала в 2001 году. С появлением C# 2.0 вместе с Visual Studio 2005 в язык были добавлены несколько важных новых свойств, включая обобщения, итераторы и безымянные методы. В версию C# 3.0 добавлены методы расширений, лямбда-выражения и, Language-Integrated Query (LINQ), позволяющее выполнять запросы к данным. Версия C# 4.0 (2010 г.) улучшила совместимость межязыкового программирования, а также вышло дополнение к среде .NET Framework, составляющие библиотеку параллельно выполняемых задач (Task Parallel Library (TPL)). В версию C# 5.0 была добавлена собственная поддержка асинхронной обработки данных на основе применения задач. Версия C# 6.0 явилась дополняющим обновлением, включающим такие дополнения, как строковая интерполяция, усовершенствованные способы реализации свойств, и многие другие улучшения.

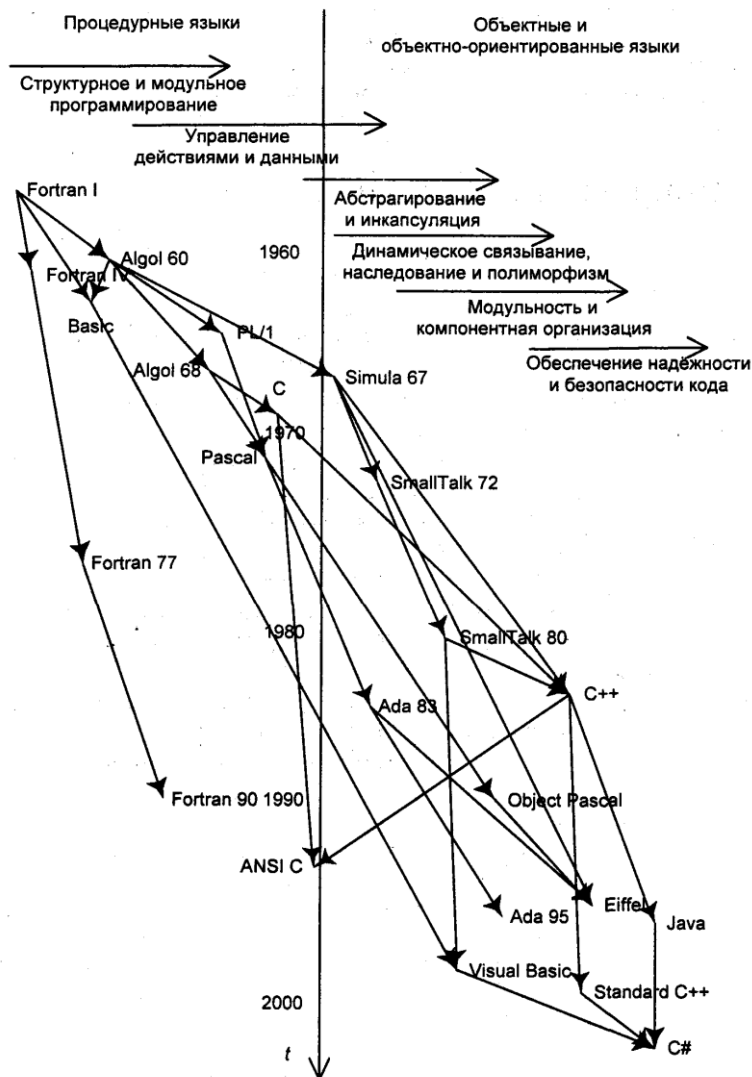


Рис. 8 – Генеалогия распространенных языков программирования и развитие основных концепций

ПРАКТИЧЕСКАЯ РАБОТА № 1

Тема: «Знакомство с интегрированной средой разработчика Visual Studio»

Цель работы – познакомиться с средой разработки приложений Microsoft Visual Studio 2017. Научится проводить установку пакетов прикладных программ специального назначения.

Теоретические сведения

1.1. Интегрированная среда разработки (IDE)

Интегрированная среда разработки (ИСР) – это система программных средств, используемая программистам для разработки программного обеспечения. В английском языке такая среда называется Integrated development environment или сокращённо IDE.

Часто ИСР также содержит средства для интеграции с системами управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя.

Многие современные среды разработки также включают окно просмотра программных классов, инспектор объектов и диаграмму иерархии классов – для использования при объектно-ориентированной разработке ПО.

Большинство современных ИСР предназначены для разработки программ на нескольких языках программирования одновременно.

Один из частных случаев ИСР – **среды визуальной разработки**, которые включают в себя возможность визуального редактирования интерфейса программы.

Основным окном, является **текстовый редактор**, который используется для ввода исходного кода в ИСР и ориентирован на работу с последовательностью символов в текстовых файлах. Такие редакторы обеспечивают расширенную функциональ-

ность – подсветку синтаксиса, сортировку строк, шаблоны, конвертацию кодировок, показ кодов символов и т. п. Иногда их называют редакторами кода, так как основное их предназначение – написание исходных кодов компьютерных программ.

Подсветка синтаксиса – выделение синтаксических конструкций текста с использованием различных цветов, шрифтов и начертаний. Обычно применяется в текстовых редакторах для облегчения чтения исходного текста, улучшения визуального восприятия. Часто применяется при публикации исходных кодов в Интернет.

Одна из наиболее важных частей ИСР – **отладчик**, который представляет собой модуль среды разработки или отдельное приложение, предназначенное для поиска ошибок в программе. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения программы, устанавливать и удалять контрольные точки или условия остановки и т. д.

Основным процессом отладки является трассировка. **Трассировка** – это процесс пошагового выполнения программы. В режиме трассировки программист видит последовательность выполнения команд и значения переменных на данном шаге выполнения программы, что позволяет легче обнаруживать ошибки.

Трассировка может быть начата и окончена в любом месте программы, выполнение программы может останавливаться на каждой команде или на точках останова, трассировка может выполняться с заходом в процедуры/функции и без заходов.

Наиболее важным модулем ИСР при совместной разработке проектов средней и высокой степени сложности является система управления версиями. **Система управления версиями** (английская аббревиатура CVS) - программное обеспечение для облегчения работы с изменяющейся информацией. Она позволяет хранить несколько версий одного и того же документа, при

необходимости, возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение и многое другое.

Такие системы наиболее широко применяются при разработке программного обеспечения, для хранения исходных кодов разрабатываемой программы.

Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов, в частности, они всё чаще применяются в САПР, обычно в составе систем управления данными об изделии.

Управление версиями используется в инструментах конфигурационного управления различных устройств и систем.

Для того, чтобы создавать автономные программы для Windows на языке C# ИСР должна содержать следующие основные составляющие:

- редактор исходного текста программ;
- дизайнер визуальных форм для создания интерфейса пользователя;
- компилятор;
- отладчик.

Большинство программистов для создания Windows приложений используют интегрированную среду Microsoft Visual Studio (имеются бесплатные версии), но существуют также и проекты с открытым исходным кодом, например, SharpDevelop и MonoDevelop.

1.2. Интегрированная среда разработчика Visual Studio.

Microsoft Visual Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Ознакомиться с документацией продуктов Microsoft можно по ссылке <https://docs.microsoft.com/ru-ru/>, а непосредственно с Visual Studio 2017 — <https://docs.microsoft.com/ru-ru/visualstudio/?view=vs-2017> (рис. 1).

В настоящее время скачать Visual Studio возможно в 3-х конфигурациях: Visual Studio Community, Visual Studio Professional, Visual Studio Enterprise (рис. 2).

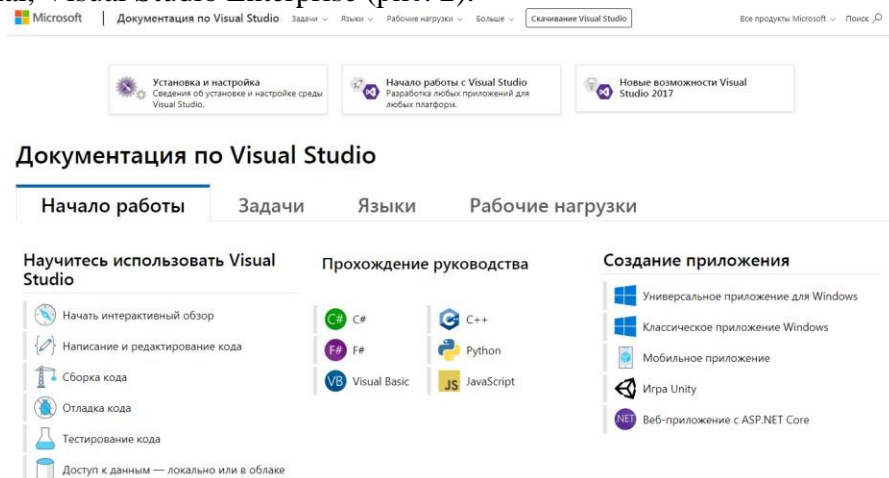


Рис. 1 — Сайт документации программы Visual Studio

Visual Studio Professional — версия, предназначенная для небольших групп программистов. Бесплатная пробная версия.

Visual Studio Enterprise — версия, предназначенная для крупных организаций. Доступна в виде бесплатной пробной версии.

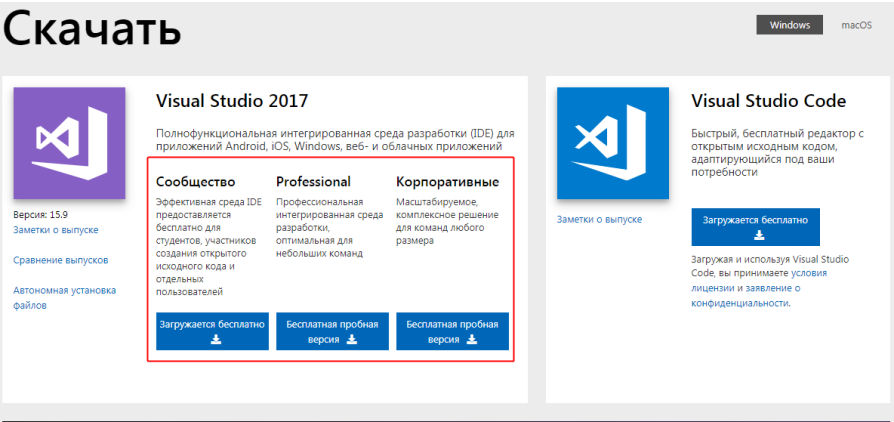


Рис. 2 — Сайт загрузки пакета программ Visual Studio

Visual Studio Community — бесплатная версия, предназначенная для студентов, участников создания открытого исходного кода и отдельных пользователей.

Данный пакет прикладных программ можно скачать как для операционной системы **Windows**, так и для **macOS** (рис. 3).

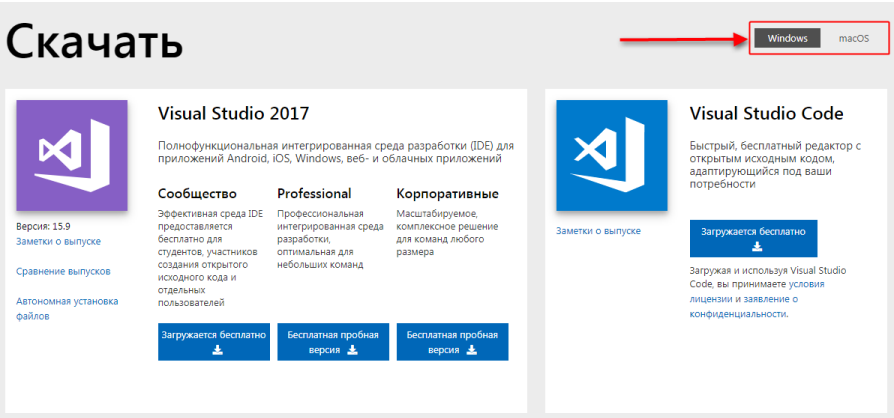


Рис. 3 — Поддерживаемые операционные системы

Перед установкой программы необходимо убедиться, что компьютер соответствует всем системным требованиям, перейдя по ссылке <https://docs.microsoft.com/ru-ru/visualstudio/productinfo/vs2017-system-requirements-vs> (рис. 4).

Фильтровать по названию

Минимальные требования к системе относятся к следующим продуктам.

- Visual Studio Enterprise 2017
- Visual Studio Professional 2017
- Visual Studio Community 2017
- Visual Studio Team Explorer 2017
- Visual Studio Test Professional 2017
- Агент тестирования Visual Studio 2017
- Контроллер тестирования Visual Studio 2017
- Интеграция Visual Studio Team Foundation Server 2017 с Office
- Visual Studio Feedback Client 2017

Поддерживаемые операционные системы

Visual Studio 2017 может устанавливаться и запускаться в следующих операционных системах:

- Windows 10 версии 1507 или выше: Домашняя, Профессиональная, для образовательных учреждений и Корпоративная (выпуски LTSC и S не поддерживаются)
- Windows Server 2016: Standard и Datacenter
- Windows 8.1 (с обновлением 2919355): Core, Профессиональная и Корпоративная
- Windows Server 2012 R2 (с обновлением 2919355): Essentials, Standard, Datacenter
- Windows 7 с пакетом обновления 1 (с последними обновлениями Windows): Домашняя расширенная, Профессиональная, Корпоративная, Максимальная

Рис. 4 — Необходимые системные требования

Также необходимо убедиться, что достаточно места на компьютере. Данная программа в среднем занимает **20-50 Гб**. Однако, если устанавливать все библиотеки, то займет около **130 Гб** (рис. 5).

Совместимость платформ

Требования к системе

Условия лицензий

Visual Studio для Mac

Visual Studio 2015

Visual Studio 2013

Visual Studio 2012

Visual Studio 2010

Ресурсы поддержки Visual Studio

Team Foundation Server 2018

Team Foundation Server 2017

Team Foundation Server 2015

Оборудование

- Процессор с тактовой частотой не ниже 1,8 ГГц. Рекомендуется использовать как минимум двухъядерный процессор.
- 2 Гб ОЗУ; рекомендуется 4 Гб ОЗУ (минимум 2,5 Гб при выполнении на виртуальной машине)
- Место на жестком диске: до 130 Гб свободного места в зависимости от установленных компонентов, обычно для установки требуется от 20 до 50 Гб свободного места
- Скорость жесткого диска: для повышения производительности установите Windows и Visual Studio на твердотельный накопитель (SSD)
- Видеоадаптер с минимальным разрешением 720p (1280 на 720 пикселей); для оптимальной работы Visual Studio рекомендуется разрешение WUXGA (1366 на 768 пикселей) или более высокое.

Поддерживаемые языки

Visual Studio доступна на следующих языках: английский, китайский (упрощенное и традиционное письмо), чешский, французский, немецкий, итальянский, японский, корейский, польский, португальский (Бразилия), русский, испанский и турецкий.

Язык Visual Studio можно выбрать во время установки. Установщик Visual Studio доступен на тех же четырнадцать языках и будет соответствовать языку Windows (если он доступен).

Рис. 5 — Требования к компьютеру

Для скачивания необходимо перейти по ссылке <https://visualstudio.microsoft.com/ru/downloads> и скачать **Visual Studio Community 2017**. После скачивания и открытия файла, появится окно установки (рис. 6).

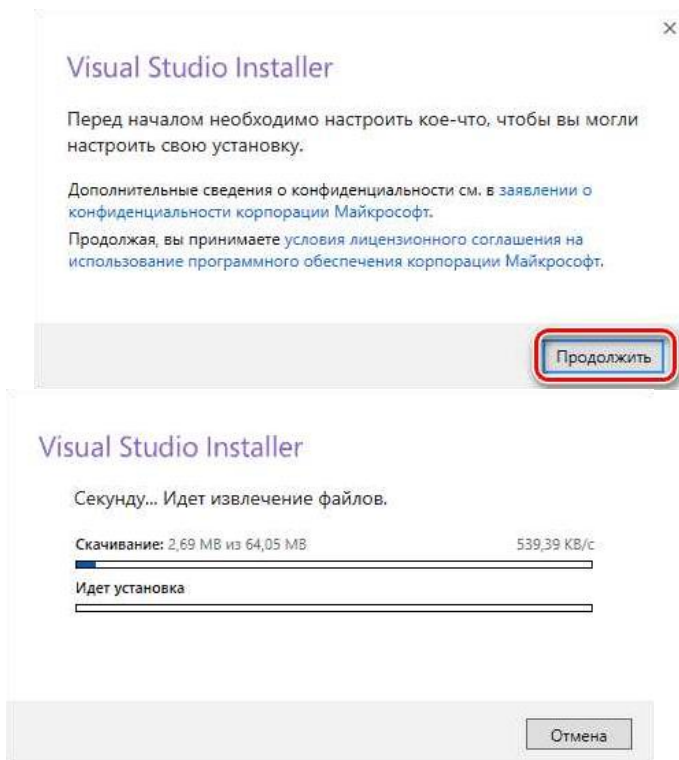


Рис. 6 — Окно установки

После нажатия «**Продолжить**», происходит скачивание основных файлов, необходимых для дальнейшей установки программы. По окончании процесса загрузки нужно будет выбрать устанавливаемые с средой компоненты (рис. 7).

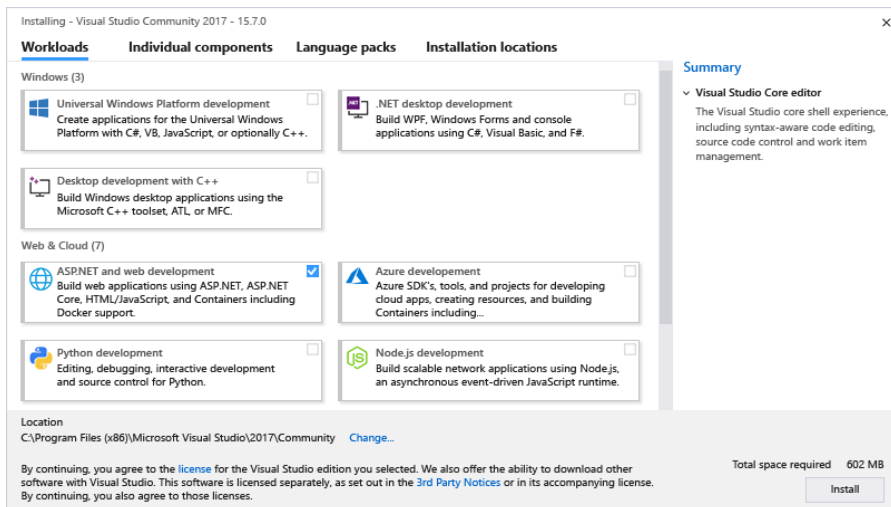


Рис. 7 — Диалоговое окно Рабочие нагрузки

Внимательно обращайтесь внимание на «Общий объем установки» (**Total space required**) и «Расположение» (**Location**), это то место, куда установится **Visual Studio**.

От диалогового окна рабочие нагрузки (см. рис. 7), будет зависеть то, какие компоненты будут установлены. По умолчанию никакая из галочек не нажата. Если не отмечать компоненты, то установится только оболочка программы. В дальнейшем ее можно будет изменять с помощью **Visual Studio Installer**. Данная подпрограмма необходима для изменения и внесения дополнений в среду разработки **Visual Studio** (добавление дополнительных компонентов, языков программирования).

Для изучения визуального программирования необходимо установить на вкладке **Workloads (Рабочие нагрузки)** флажки выбора напротив следующих компонентов:

- Universal Windows Platform development (Разработка приложений для универсальной платформы Windows);
- .NET desktop development (Разработка классических при-

ложений .NET);

В целом, можно выделить все представленные средства разработки, но одновременная установка всех представленных компонентов может сильно отразиться на производительности программы. С полной документацией по рабочим нагрузкам в Visual Studio 2017 можно ознакомиться здесь <https://visualstudio.microsoft.com/ru/vs/visual-studio-workloads/>.

На вкладке Individual components (Отдельные компоненты) для полноценной работы можно установить (рис. 8) флажки у следующих компонентов:

- Class Designer (Конструктор классов);
- GitHub extension for Visual Studio (Расширение GitHub для Visual Studio);
- PowerShell tools (Инструменты PowerShell).

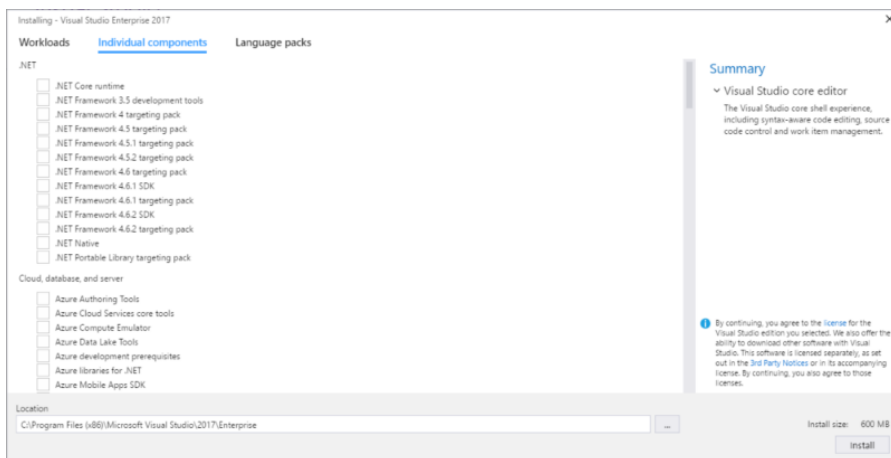


Рис. 8 — Диалоговое окно Отдельные компоненты

На странице Language packs (рис. 9) можно выбрать язык интерфейса.

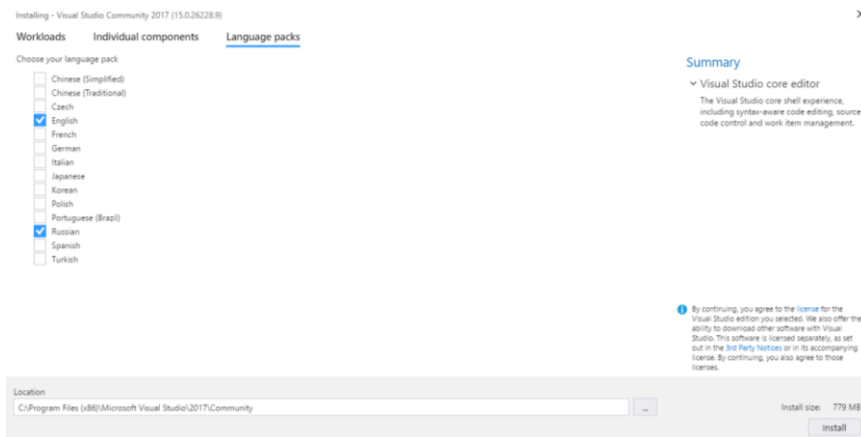


Рис. 9 — Диалоговое окно Language packs

После выбора конфигурации устанавливаемой программы, нажмите кнопку **Install** (Установить) и подождите, пока выбранные компоненты не будут установлены.

После начинается процесс загрузки (рис. 10). После завершения процесса установки нажмите кнопку **Launch** (Запустить).

В поиске программ операционной системы Windows необходимо найти программу Visual Studio и запустить ее.

После первого запуска Visual Studio 2017, появится окно с запросом авторизации. Если вы ранее зарегистрировали учетную запись Microsoft, то можно воспользоваться регистрационными данными такого аккаунта. В случае отсутствия учетной записи зарегистрируйте новую, перейдя по ссылке <https://signup.live.com/> (рис. 11).

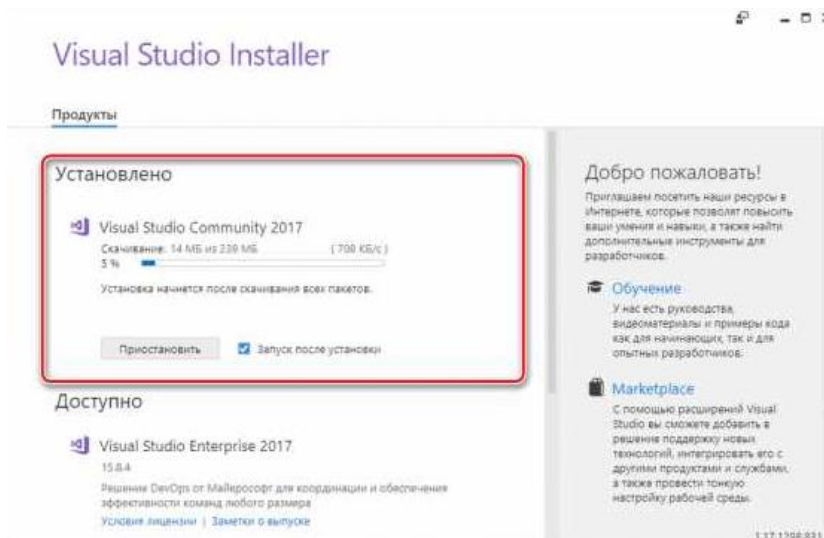


Рис. 10 — Этап загрузки Visual Studio

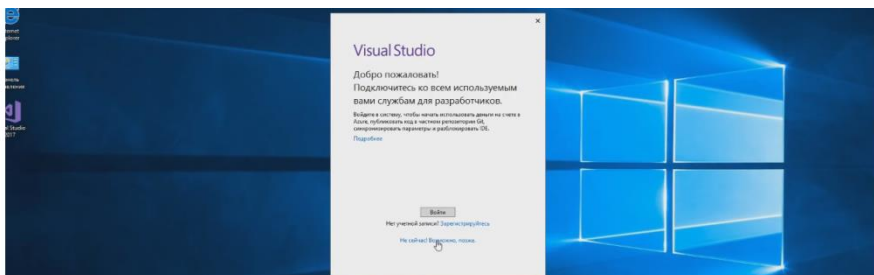


Рис. 11 — Окно приветствия Visual Studio

Если указать **«Не сейчас! Возможно, позже»**, аккаунт можно создать в другое время и продолжить работать с средой разработки.

Так же, при первом запуске Visual Studio 2017 откроется окно с предложением настроить среду под свои предпочтения. В раскрывающемся списке Development Settings (Настройки разработки) выберите пункт Visual C# и цветовую схему (рис. 12).

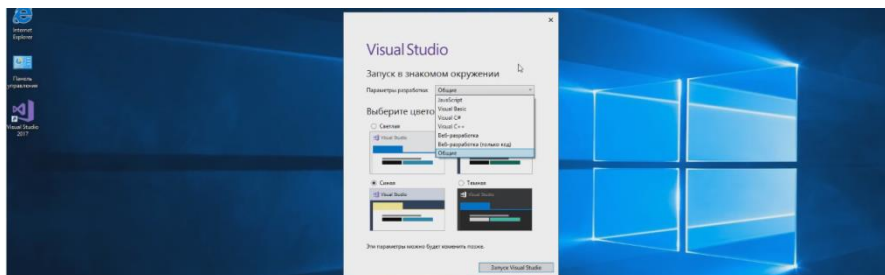


Рис. 12 — Диалоговое окно выбора темы

После запуска **Visual Studio** появляется начальная страница (рис. 13) со списком последних проектов, а также командами «Создать проект» и «Открыть проект». Нажмите ссылку «Создать проект» или выберите в меню «Файл» команду «Создать проект», на экране появится диалог для создания нового проекта.

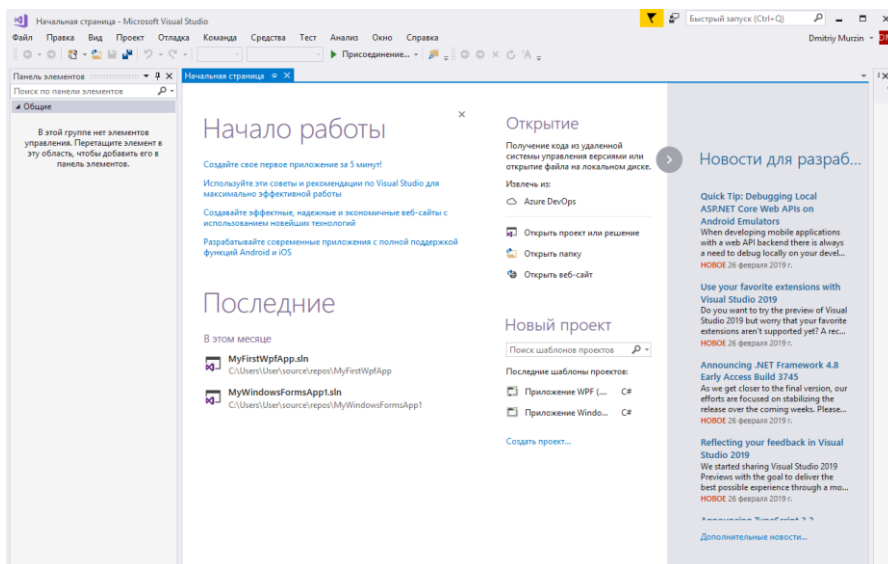


Рис. 13 — Домашняя страница Visual Studio

В окне «Создания проекта» необходимо выбрать в какой среде будут выполняться работа системы. Для этого выберите Visual C# в левой области диалогового окна Новый проект, а в списке шаблонов выберите Приложение Windows Forms (.NET Framework) (рис. 14).

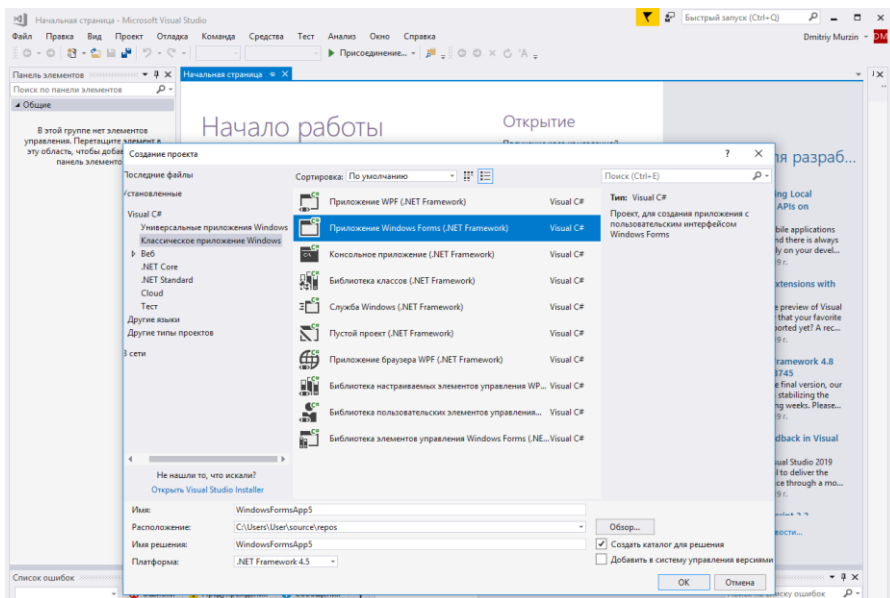


Рис. 14 — Окно «Создание проекта»

После создания проекта, откроется рабочая область, с автоматически созданной на ней формой (рис. 15). Как и большинство других классических Windows-приложений, интерфейс Visual Studio содержит строку меню, панель инструментов с часто вызываемыми командами и строку состояния внизу окна. В правой части окна программы расположена панель Solution

Explorer (Обозреватель решений) со списком открытых проектов.

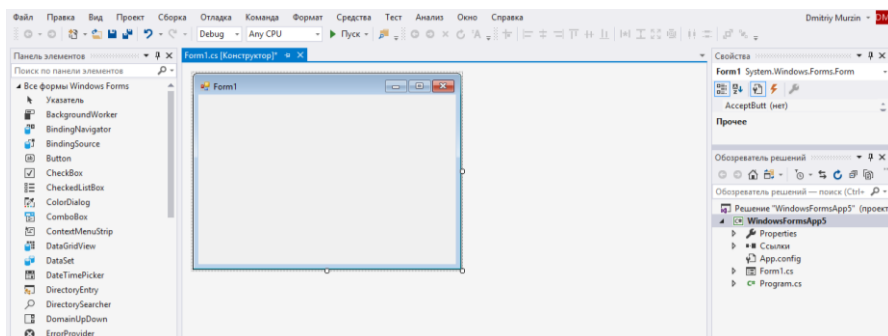


Рис. 15 — Рабочая область

Слева от рабочей области есть несколько вкладок, например «Панель элементов» (рис. 15). Путем перемещения на форму элементов можно создавать новые элементы визуального интерфейса.

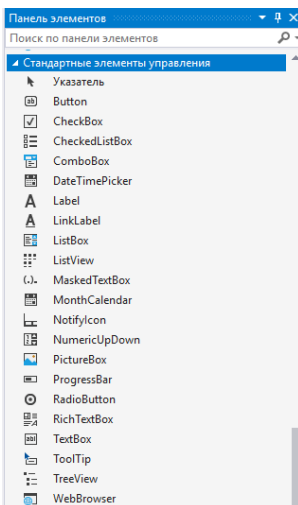


Рис. 15 — Вкладка панель инструментов

1.3. Интегрированная среда разработчика Sharp Develop

Альтернативной средой для разработки программ на C# является среда SharpDevelop, обладающая богатым функционалом, скоростью работы и занимает мало места.

Бесплатная интегрированная среда разработка SharpSevelor 5 работает на Microsoft Windows Vista, 7, 8 и старше.

Для операционной системы Windows XP, необходимо установить SharpSevelor версии 4.

Для установки среды перейдите по ссылке <http://www.icsharpcode.net/OpenSource/SD/Download/Default.aspx> (рис. 16) и скачайте последнюю версию среды SharpDevelop и установите её.

Download Our Most Current Builds

Downloads on This Page

- [Downloads for SharpDevelop 5](#) (Frameworks 2.0 to 4.5.1)
- [Downloads for SharpDevelop 4.4](#) (Frameworks 2.0 to 4.5.1)
- [Downloads for SharpDevelop 3.2](#) (Frameworks 2.0, 3.0 and 3.5)
- [Downloads for SharpDevelop 2.2](#) (built for .NET Framework 2.0, unsupported)
- [Downloads for SharpDevelop 1.1](#) (built for .NET Framework 1.1, unsupported)

Downloads for SharpDevelop 5 (C# support only!)

- Setup [Download](#) [13511 KB]
- Source code [Download](#) [42901 KB]
- Xcopyable "Installation" [Download](#) [16280 KB]

Version information: *SharpDevelop 5.1, 4/14/2016*

SharpDevelop 5.x can take advantage of the following software if you install it:

- [Microsoft .NET Framework 4.5.1 Developer Pack](#) for .NET 4.5 code completion documentation
- [Microsoft Windows SDK for Windows 7 and .NET Framework 4](#) (strongly recommended!)
- [Microsoft F#](#) for F# support
- [StyleCop](#) for source analysis support.
- [TortoiseGit](#) for Git source control support.
- [TortoiseSVN](#) for Subversion source control support.
- [SHFB](#) for documentation generation support.
- [WiX](#) for building installers.

Рис. 16 – Настройка SharpDevelop

Для русификации интерфейса программы в меню «Tools» - «Options» выберите русскую локализацию (рис. 17). Проект создается подобно проекту в Visual Studio.

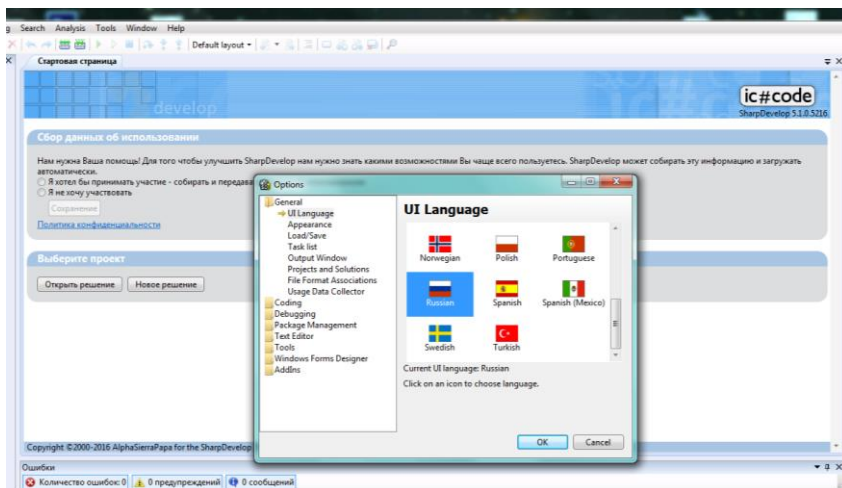


Рис. 17 – Настройка SharpDevelop

Порядок выполнения практической работы

- 1) Скачайте среду разработки Visual Studio.
- 2) Зарегистрируйтесь и/или войдите в аккаунт разработчика.
- 3) Установите среду разработки Visual Studio.
- 4) Настройте основные компоненты Visual Studio для работы с Windows Forms и WPF.
- 5) Запустить программу Visual Studio.
- 6) Создайте проект Windows Forms и сохраните его на диск.
- 7) Установите и настройте среду SharpDevelop.
- 8) Оформите отчет согласно выполненным этапам установки среды разработки (для создания скриншота используйте кнопку Print Screen).

Контрольные вопросы

- 1) Какие варианты установки Visual Studio 2017?
- 2) Перечислите и опишите основные этапы установки интегрированной среды Visual Studio.

- 3) Какие основные приемы настройки интегрированной среды Visual Studio?
- 4) Какие основные требования к операционной системе и компьютерному оснащению для установки Visual Studio?
- 5) Какие необходимо выбрать компоненты установки Visual Studio для создания Windows приложений?

ПРАКТИЧЕСКАЯ РАБОТА № 2

Тема: «Изучение процесса создания проекта на языке C#. Компиляция и отладка приложения.»

Цель работы – научиться создавать простое приложение в среде Visual Studio, делать отладку программы, производить обработку событий в форме.

Теоретические сведения

2.1. Краткие сведения о .NET Framework и C#

Net Framework – это технология, разработанная Microsoft, которая упрощает написание программ для операционных систем, мобильных устройств, сайтов и других разработок Microsoft. Кроме среды выполнения программы в Framework существуют библиотеки классов, которые упрощают разработку безопасных Windows приложений и приложений на основе Интернет-технологий (рис.2.1).

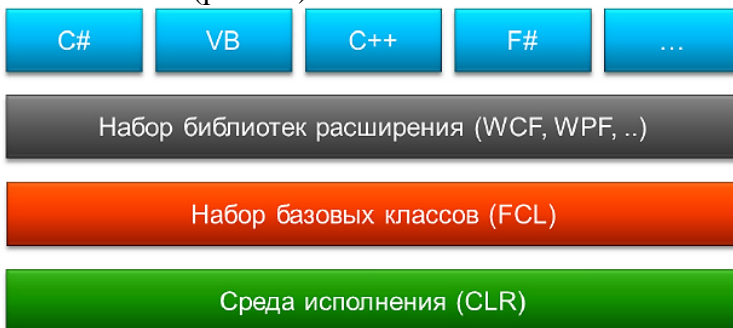


Рис. 2.1 – Структура .NET Framework

В состав платформы .NET входит библиотека классов *Framework Class Library* (FCL). Частью этой библиотеки является базовый набор классов *Base Class Library* (BCL), в который входят классы для работы со строками и коллекциями данных, для поддержки многопоточности, и множество других классов.

В FCL также входят компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML и базами данных, для создания пользовательских интерфейсов.

В стандартную поставку платформы .NET включено несколько компиляторов. Это компиляторы языков C#, F#, Visual Basic .NET, C++/CLI. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сторонними производителями. Имена элементов библиотеки FCL не зависят от языка программирования. Специфичной частью языка остаётся только синтаксис. Этот факт упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой.

Для поддержки межъязыкового взаимодействия служат две спецификации платформы .NET. *Общая система типов* (Common Type System, CTS) описывает набор типов, который должен поддерживаться любым языком программирования для .NET. *Общезыковая спецификация* (Common Language Specification, CLS) – это общие правила поведения для всех .NET-языков.

C# является специально разработанным языком поддержки .Net Framework. Хотя под .Net Framework можно программировать и на других языках, в C# реализована полная поддержка этой технологии.

2.1.2 Компиляция и язык MSIL

.NET-приложения компилируются фактически в два этапа. На первом этапе исходный код компилируется во время построения проекта и вместо исполняемого файла с машинными кодами получается сборка (assembly), содержащая команды промежуточного языка MSIL (Microsoft Intermediate Language — промежуточный язык Microsoft). Код MSIL сохраняется в файле на диске. При этом файлы MSIL (сокращенно IL), генерируемые компилятором, например, C#, идентичны IL-файлам, генериру-

емым компиляторами с других языков .NET. В этом смысле платформа остается в неведении относительно языка. Самой важной характеристикой среды CLR (общезыковой средой выполнения) является то, что она общая; одна среда выполняет как программы, написанные на C#, так и программы на языке VB.NET.

Второй этап компиляции наступает непосредственно перед фактическим выполнением страницы. На этом этапе CLR транслирует промежуточный код IL в низкоуровневый собственный машинный код, выполняемый процессором (рис. 2.2).

Это позволяет обеспечить перенос программы с одной платформы на другую, а также дополнительную защиту от ошибок и ряд других преимуществ. Правда, с небольшой потерей в производительности.

Управляемый код – это код, который выполняется в CLR. В C# есть возможность выйти за рамки управляемого кода, если важны критерии производительности или есть другие потребности при написании программы.

При выполнении .NET-программы системы CLR активизирует JIT-компилятор, который затем превращает MSIL во внутренний код процессора. Этот этап известен как оперативная компиляция "на лету" (Just-In-Time) или JIT-компиляция, и он проходит одинаково для всех приложений .NET. При исполнении программы CLR берет на себя управление памятью, контроль типов и решает за приложение ряд других задач.

Стандартный JIT-компилятор работает по запросу. Когда вызывается тот или иной метод, JIT-компилятор анализирует IL-код и производит высокоэффективный машинный код, выполняемый очень быстро. JIT-компилятор достаточно интеллектуален, чтобы распознать, был ли код уже скомпилирован, поэтому во время выполнения программы компиляция происходит лишь при необходимости. По ходу своего выполнения .NET-

программа работает все быстрее и быстрее, так как повторно используется уже скомпилированный код.

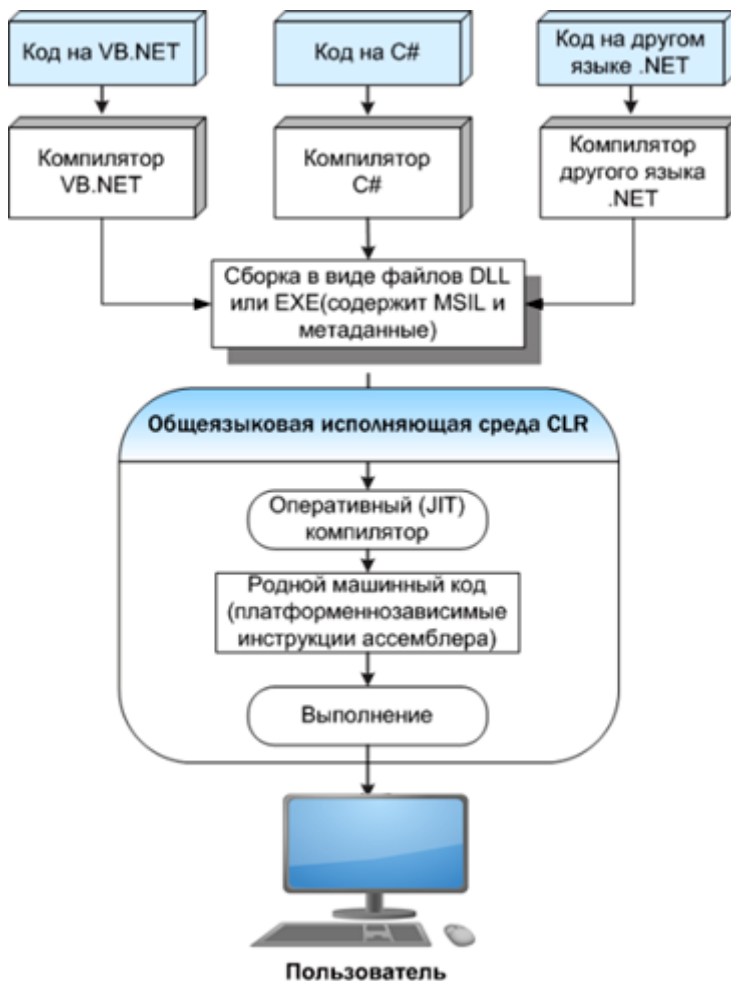


Рис. 2.2 – Схема компиляции .NET-приложения

Спецификация CLS подразумевает, что все языки платформы .NET генерируют очень похожий IL-код. Кроме того, при компилировании программы в дополнение к MSIL формируется еще один компонент — метаданные. Они описывают данные, используемые программой, и это позволяет коду взаимодействовать с другим кодом. В результате объекты, созданные на одном языке, доступны и могут наследоваться на другом. То есть можно создать базовый класс на языке VB.NET, а производный от него класс — на языке C#.

2.1.3. Структура первого приложения

Для написания программы необходимо создать проект. *Проект* содержит все исходные материалы для приложения (файлы исходного кода, ресурсов, значки, ссылки на внешние файлы, данные конфигурации). Более глобальное понятие — *решение* (*solution*) содержит один или несколько проектов, один из которых может быть указан как стартовый проект. Выполнение решения начинается со стартового проекта.

При создании простейшей C# программы создается папка решения, в которой для каждого проекта создается подпапка проекта, а уже в ней будут создаваться другие подпапки с результатами компиляции приложения.

При создании проекта можно выбрать его тип, а среда разработки создаст каркас проекта в соответствии с выбранным типом (для оконных приложений используйте тип *Приложения Windows Forms*). Проект в среде разработки состоит из файла проекта (расширение .csproj), одного или нескольких файлов исходного текста (с расширением .cs), файлов с описанием окон формы (с расширением .designer.cs), файлов ресурсов (с расширением .resx), а также ряда служебных файлах.

В *файле проекта* находится информация о модулях, составляющих данный проект, входящих в него ресурсах, а также параметров построения программы. Файл проекта создается авто-

матически и не предназначен для ручного редактирования. *Файл исходного текста* – программный модуль предназначен для размещения текстов программ. В этом файле размещается текст программы. Модуль имеет следующую структуру:

```
// Раздел подключенных пространств имен
using System;
using System.Windows.Forms;
// Пространство имен проекта
namespace MyFirstApp
{
    // Класс окна
    public partial class MainForm : Form
    {
        // Методы окна
        public MainForm()
        {
            InitializeComponent();
        }
    }
}
```

В разделе пространства имен (после using) описываются используемые в программе пространства имён, которые включают в себя классы, выполняющие определённые функции. Так System.Windows.Forms содержит классы пользовательского интерфейса, которые предоставляет в операционная система Microsoft Windows. Для работы с сетью используется System.Net, для работы с файлами – в System.IO и т.д. (подробнее на официальном сайте [https://msdn.microsoft.com/ru-ru/library/mt472912\(v=vs.110\).aspx#Пространства имен](https://msdn.microsoft.com/ru-ru/library/mt472912(v=vs.110).aspx#Пространства_имен)).

Для предотвращения конфликтов имён классов и переменных, классы нового проекта также размещены в отдельное пространство имен, которое определяется ключевым словом namespace, после которого следует имя пространства.

Внутри пространства имен помещаются классы – в новом проекте это класс окна, который содержит все методы для

управления поведением окна. В примере в определении класса присутствует ключевое слово `partial`, информирующее компилятор, что в исходном тексте представлена только часть класса, а служебные методы для обслуживания окна скрыты в другом месте. Внутри класса располагаются переменные, методы и другие элементы программы. Фактически, основная часть программы размещается внутри класса при создании обработчиков событий. После компиляции программы среда разработки создает исполняемые `.exe`-файлы в каталоге `bin`.

Для сохранения проекта в среде разработки используется команда: *Файл* → *Сохранить всё* (`Ctrl+Shift+S`). Для открытия существующего проекта используется команда *Файл* → *Открыть проект* (`Ctrl+Shift+O`), либо можно найти в папке файл проекта с расширением `.sln` и сделать на нём двойной щелчок.

2.2. Общие концепции синтаксиса языка C#

Специально для платформы .NET был разработан новый язык программирования C#, который сочетает простой синтаксис, похожий на синтаксис языков C, C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. При разработке языка использовалась концепция безопасного программирования, нацеленная на создание надёжного и простого в сопровождении кода.

Основными понятиями в языке C# являются *программы*, *сборки*, *пространства имён*, *пользовательские типы*, *элементы типов*. Исходный код программы на языке C# размещается в одном или нескольких текстовых файлах, имеющих расширение `*.cs`. В программе объявляются пользовательские типы, такие как, классы и структуры, состоящие из элементов, например, метод класса. Типы могут быть логически сгруппированы в пространства имён, а физически (после компиляции) – в сборки, представляющие собой файлы с расширением `.exe` или `.dll`.

Исходный код программы на языке С# – это набор *операторов* (statements), *директив препроцессора* и *комментариев*.

Комментарии бывают двух видов:

1) *однострочный комментарий* – это комментарий, начинающийся с последовательности `//` и продолжающийся до конца строки;

2) *блочный (многострочный) комментарий* – все символы, заключённые между парами `/*` и `*/`.

В С# различаются строчные и прописные символы при записи идентификаторов и ключевых слов. При кодировании операторы одного уровня вложенности обычно сопровождаются одинаковым начальным отступом.

2.2.1. Описание данных в С#

Основой платформы .NET является развитая система типов (таблица 2.1). Подробная таблица представлена в **Приложении А**. С# является строго типизированным языком. Это означает то, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Такая строгая проверка типов позволяет предотвратить ошибки и повысить надежность программ.

CLR использует уникальное имя типа, составляющей которого является указание на пространство имён. Так, для представления строк служит тип `System.String`, где `System` – название пространства имён. Для наиболее распространённых типов платформы .NET язык С# предлагает короткие имена-псевдонимы. Например, тип `int` в С# – это псевдоним типа `System.Int32`, тип `string` – псевдоним типа `System.String`.

Все типы платформы .NET можно разделить на *типы значений* (value types) и *ссылочные типы* (reference types). Переменная типа значения непосредственно содержит данные. К типам значений относятся *структуры* и *перечисления*.

Таблица 2.1 – Типы данных C#

| Логический тип | | | |
|---------------------------------------|-----------------------------|--|--------------------|
| Имя типа | Имя типа в CLR | Значения | Размер |
| <code>bool</code> | <code>System.Boolean</code> | true, false | 8 бит |
| Арифметические целочисленные типы | | | |
| Имя типа | Системный тип | Диапазон | Размер |
| <code>byte</code> | <code>System.Byte</code> | 0 – 255 | Беззнаковое, 8 бит |
| <code>short</code> | <code>System.Int16</code> | –32768 ... – 32767 | Знаковое, 16 бит |
| <code>int</code> | <code>System.Int32</code> | $\approx (-2 \cdot 10^9 \dots -2 \cdot 10^9)$ | Знаковое, 32 бит |
| <code>long</code> | <code>System.Int64</code> | $\approx (-9 \cdot 10^{18} \dots 9 \cdot 10^{18})$ | Знаковое, 64 бит |
| Арифметический тип с плавающей точкой | | | |
| Имя типа | Имя типа в CLR | Диапазон | Точность |
| <code>float</code> | <code>System.Single</code> | $-1.5 \cdot 10^{-45} \dots +3.4 \cdot 10^{38}$ | 7 цифр |
| <code>double</code> | <code>System.Double</code> | $-5.0 \cdot 10^{-324} \dots +1.7 \cdot 10^{308}$ | 15–16 цифр |

Структуры включают *пользовательские структуры*, *простые типы* (это *числовые типы* и тип `bool`) и *типы с поддержкой null*. Переменная ссылочного типа, далее называемая *объектом*, хранит ссылку на данные, которые размещены в управляемой динамической памяти. Ссылочные типы – это *классы*, *интерфейсы*, *строки*, *массивы*, *делегаты* и тип `object`. С точки

зрения компилятора языка С# типы делятся на *примитивные типы* и *пользовательские типы*.

Поддержка примитивных типов обеспечена компилятором, такие типы не нуждаются в дополнительном объявлении. Простые типы и их варианты с поддержкой `null`, а также типы `string`, `object` и `dynamic` принято относить к примитивным типам. Пользовательские типы перед применением должны быть описаны при помощи специальных синтаксических конструкций.

Числовые типы делятся на *целочисленные типы*, *типы с плавающей точкой* (удовлетворяют стандарту IEEE 754) и *тип decimal* (значение (96 бит) и экспонента, где экспонента (7 бит) — степень десятки).

При использовании нескольких языков программирования не следует использовать типы `sbyte`, `ushort`, `uint`, `ulong`, так как они не соответствуют Common Language Specification.

В языке С# необходимо соблюдать соответствие типов при присваивании и вызове методов. В случае несоответствия выполняется *преобразование типов*, которое бывает явным и неявным. Для *явного преобразования* (explicit conversion) служит операция приведения в форме (*целевой-тип*) *выражение*. *Неявное преобразование* (implicit conversion) не требует особых синтаксических конструкций и осуществляется компилятором. Неявное преобразование безопасно и определено для чисел A и B, если на рис. 2.1 существует путь из узла A в узел B.

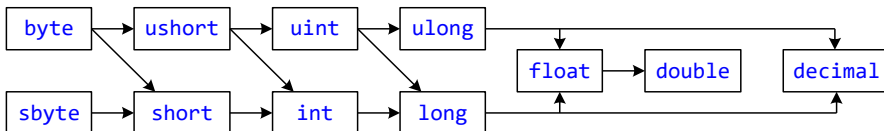


Рис. 2.1 – Схема неявного преобразования числовых типов.

Тип `char` представляет символ в Unicode-кодировке UTF-16. Тип `char` преобразуется в типы `sbyte`, `short`, `byte` явно, а в осталь-

ные числовые типы – неявно. Преобразование числового типа в тип `char` может быть выполнено только в явной форме.

Тип `bool` служит для хранения логических (булевых) значений. Переменные данного типа могут принимать значения `true` или `false`. Ни неявное, ни явное преобразование `bool` в числовые типы и обратно невозможно.

Тип `string` используется для работы со строками и является последовательностью Unicode-символов.

Тип `object` – это ссылочный тип, переменной которого можно присвоить любое значение.

Пользовательские типы обладают следующей функциональностью:

- 1) Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм;
- 2) Структура – тип значения, обеспечивающий инкапсуляцию данных, но не поддерживающий наследование (синтаксически, структура похожа на класс);
- 3) Интерфейс – абстрактный тип, реализуемый классами и структурами для обеспечения оговорённой функциональности;
- 4) Массив – пользовательский тип для представления упорядоченного набора значений;
- 5) Перечисление – тип, содержащий в качестве членов именованные целочисленные константы;
- 6) Делегат – тип, инкапсулирующий метод.

2.2.2. Идентификаторы, ключевые слова и литералы в C#

Идентификатор – это пользовательское имя для переменной, константы, метода или типа¹. При записи идентификатора

¹ произвольная последовательность букв, цифр и символов подчёркивания, начинающаяся с буквы, символа подчёркивания, либо символа @

возможно использовать четыре шестнадцатеричных цифры кода UTF-16 с префиксом `\u`. Идентификатор не может совпадать с ключевым словом языка². Примеры допустимых идентификаторов: `Temp`, `_variable`, `_` (символ подчёркивания), `@class` (используется префикс `@`, так как `class` – ключевое слово), `cl\u0061ss` (применяется код UTF-16 для символа `a`, этот идентификатор совпадает с идентификатором `@class`). Идентификатор должен быть уникальным внутри области видимости.

Область видимости переменной – это участок программы, в пределах которого переменной можно воспользоваться. В C# переменную можно объявлять в любой точке процедурного блока, при этом область ее видимости распространяется от точки объявления до конца процедурного блока.

Ключевые слова – зарезервированные идентификаторы (Приложение А), имеющие специальные значения для компилятора, и их нельзя использовать в качестве идентификаторов. В C# есть *контекстные ключевые слова*, которые имеют особое значение в ограниченном программном контексте (Приложение Б).

Литерал – это последовательность символов, которая может интерпретироваться как значение определённого типа.

Для ссылочных типов определён литерал `null`, который указывает на неинициализированную ссылку. В языке C# два булевых литерала: `true` и `false`. Целочисленные литералы могут быть записаны в десятичной или шестнадцатеричной форме³. Тип целочисленного литерала определяется следующим образом:

- Если литерал не имеет суффикса, то его тип – это первый из типов `int`, `uint`, `long`, `ulong`, который вмещает значение литерала.
- Если литерал имеет суффикс `U` или `u`, его тип – это первый из типов `uint`, `ulong`, который способен вместить значение литерала.

² за исключением того случая, когда используется специальный префикс `@`

³ Признаком шестнадцатеричного литерала является префикс `0x` (или `0X`).

- Если литерал имеет суффикс L или l⁴, то его тип – это первый из типов **long**, **ulong**, который вмещает значение литерала.
- Если литерал имеет суффикс UL, Ul, uL, ul, LU, Lu, lU, lu, его тип – **ulong**.

Если в числе с десятичной точкой не указан суффикс, то подразумевается тип **double**. Суффикс f (или F) указывает на тип **float**, суффикс d (или D) указывает на тип **double**, суффикс m (или M) определяет тип **decimal**. Число с плавающей точкой может быть записано в научном формате: 3.5E-6, -7e10, .6E+7.

Символьный литерал записывают в одинарных кавычках. Обычно это единичный символ (например, 'a'). Допустимо указать четыре шестнадцатеричных цифры кода UTF-16 с префиксом \u ("u005C" – это символ '\'). Можно использовать префикс \x и не более четырёх шестнадцатеричных цифр кода UTF-16 ("\x5c" – это тоже символ '\'). Кроме этого, для представления некоторых специальных символов используются следующие пары:

| | | | |
|----|------------------------|----|----------------------------|
| \' | – одинарная кавычка | \f | – новая страница |
| \" | – двойная кавычка | \n | – новая строка |
| \\ | – обратная косая черта | \r | – возврат каретки |
| \0 | – символ с кодом ноль | \t | – горизонтальная табуляция |
| \a | – звуковой сигнал | \v | – вертикальная табуляция |
| \b | – забой | | |

Для строковых литералов в языке C# существуют две формы. Обычно строковый литерал записывается как последовательность символов в двойных кавычках ("This is a \t tabbed string"). *Дословная форма* (verbatim form) строкового литерала – это запись строки в кавычках с использованием префикса @ (@ "There is \t no tab").

⁴ не рекомендуется, так как похож на единицу

2.3. Запуск и отладка программы в .NET Framework

Для запуска программы в среде программирования Visual Studio необходимо запустить программу выбрав в меню *Отладка* команду *начать отладку*. При этом происходит трансляция и, если нет ошибок, компоновка программы и создание единого загружаемого файла с расширением `.exe`. По ссылке <https://docs.microsoft.com/ru-ru/visualstudio/ide/find-and-fix-code-errors?view=vs-2017> можно ознакомиться с документацией по теме «Работа с кодом в Visual Studio».

Чтобы запустить приложение с подключенным отладчиком, нажмите клавишу F5 выберите *Отладка* → *начать отладку*, или выберите зеленую стрелку на панели инструментов Visual Studio (рис. 2.3).

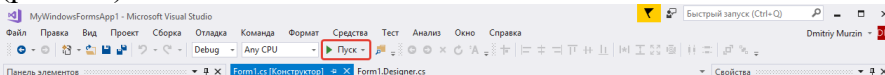


Рис. 2.3 – Меню запуска отладки .NET-приложения

Во время отладки, выделение желтым цветом показаны строки кода, который будет выполняться «Далее» (рис. 2.4).

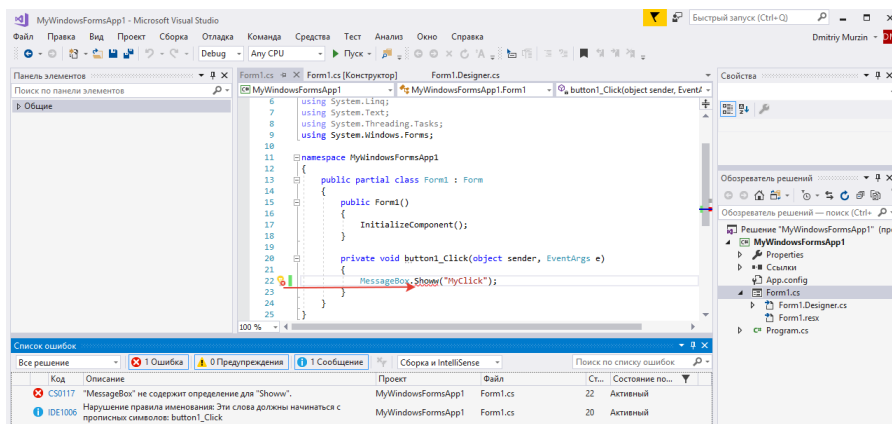


Рис. 2.4 – Процесс отладки .NET-приложения

Если в программе обнаружены ошибки, то в окне Список ошибок появятся найденные ошибки. В настройках Visual Studio можно установить опцию показа окна ошибок и запретить запуск при наличии ошибок (рис. 2.5 и 2.6).

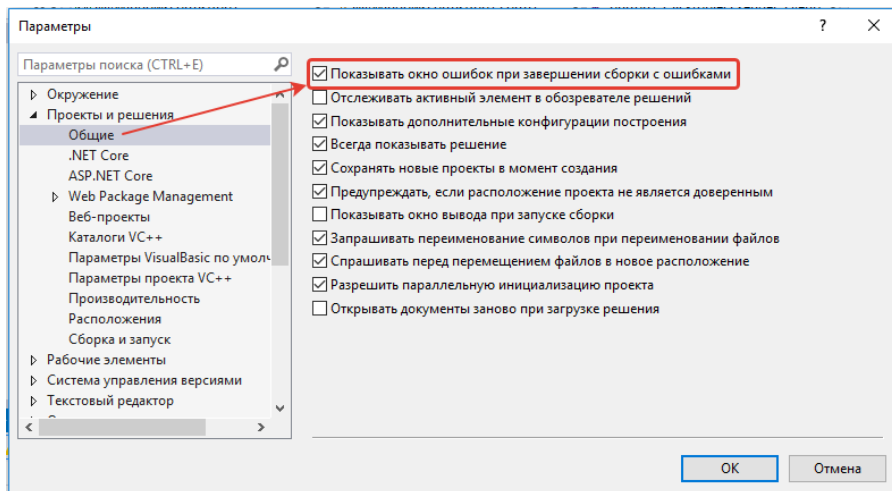


Рис. 2.5 – Настройка показа сообщений об ошибках Visual Studio

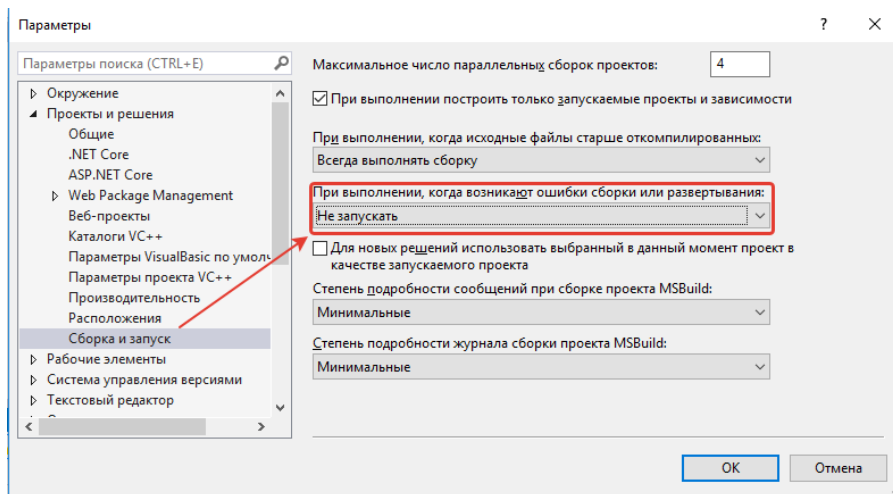


Рис. 2.6 – Настройка отключения запуска предыдущей версии при ошибках в MS Visual Studio

Практически в каждой вновь написанной программе после запуска обнаруживаются ошибки. Ошибки первого уровня (ошибки компиляции) связаны с неправильной записью операторов (орфографические, синтаксические). При обнаружении ошибок компилятор формирует список, который отображается по завершению компиляции (рис. 2.7). При выявлении ошибок компиляции в нижней части экрана появляется текстовое окно, содержащее сведения обо всех ошибках, найденных в проекте. Каждая строка этого окна содержит имя файла, в котором найдена ошибка, номер строки с ошибкой и характер ошибки. Для быстрого перехода к интересующей ошибке необходимо дважды щелкнуть на строке с ее описанием. Следует обратить внимание на то, что одна ошибка может повлечь за собой другие, которые исчезнут при ее исправлении. Поэтому следует исправлять ошибки последовательно, сверху вниз и, после исправления каждой ошибки компилировать программу снова.

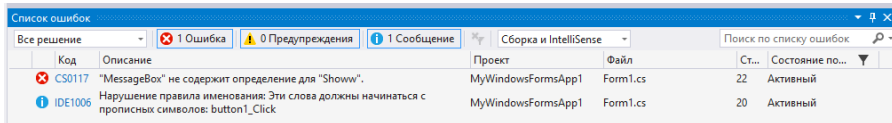


Рис. 2.7 – Окно со списком ошибок компиляции

Ошибки второго уровня (ошибки выполнения) связаны с ошибками выбранного алгоритма решения или с неправильной программной реализацией алгоритма (неверный результат расчета, переполнение, деление на ноль и др.). Поэтому перед использованием отлаженной программы ее надо протестировать с помощью набора подготовленных тестовых данных для расчета. Для выявления подобных ошибок рекомендуется установить точку останова перед подозрительным участком кода, которая устанавливается в окне редактирования кода, если кликнуть слева от нужной строки (рис. 2.8).

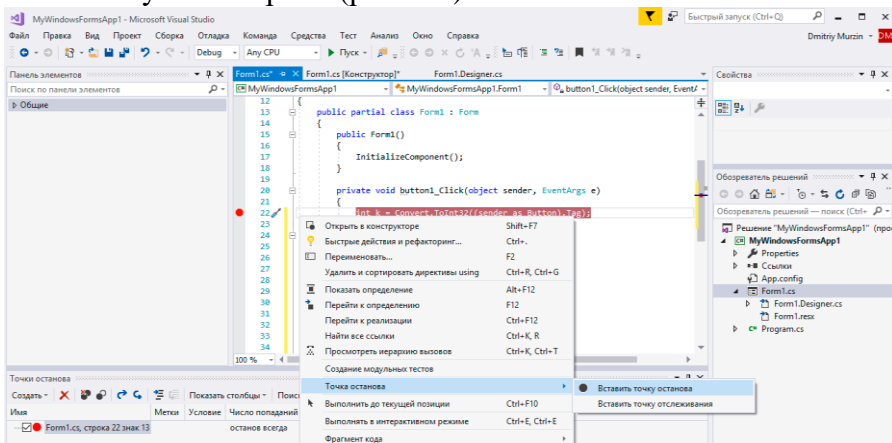


Рис. 2.8 – Фрагмент кода с точкой останова

При достижении установленной точки, программа будет остановлена, и далее код будет выполняться по шагам с помо-

щью команд *Отладка* → *Шаг с обходом* (без захода в методы) или *Отладка* → *Шаг с заходом* (с заходом в методы) (рис 2.9).

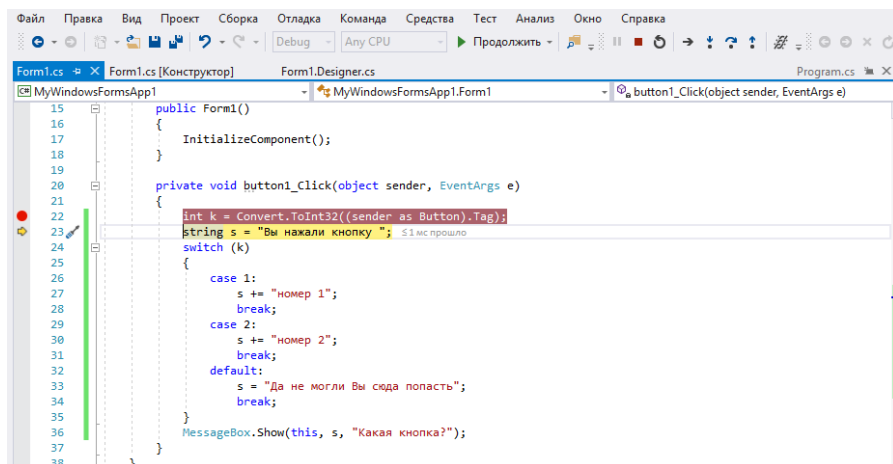


Рис. 2.9 – Отладка программы

Желтым цветом выделяется оператор, который будет выполнен. Значение переменных во время выполнения можно увидеть, наведя на них курсор. Для прекращения отладки и остановки программы необходимо выполнить команду меню *Отладка* → *Остановить отладку*. Для поиска алгоритмических ошибок можно контролировать значения промежуточных переменных на каждом шаге выполнения подозрительного кода и сравнивать их с результатами, полученными вручную.

2.4 Введение в визуальное программирование

Визуальное программирование – способ создания программы для с помощью манипулирования графическими видимыми объектами вместо написания программного кода. Основным назначением визуальных методов программирования является разработка графических интерфейсов пользователя приложения.

Средства для организации взаимодействия с пользователем, например, окна, кнопки, меню и другие элементы управления, называют графическим интерфейсом пользователя (Graphical User Interface, GUI).

Среда быстрой разработки визуального интерфейса берет на себя рутинную работу, связанную с подготовкой программы к работе, автоматически генерирует соответствующий программный код и позволяют сосредоточиться на логике работы будущей программы.

Процесс создания Windows-приложения состоит из двух основных этапов:

1) визуальное проектирование – состоит в выборе и размещении элементов управления, таких, как кнопки, переключатели, значки и т.д. на плоскости окна будущего приложения;

2) определение поведения приложения путем написания процедур обработки различных событий:

- при щелчке мышью на той или иной кнопке;
- при выборе определенного пункта меню;
- по прошествии определенного интервала времени;
- при наступлении какого-либо иного события⁵.

Такое программирование визуального интерфейса и программного кода называют событийно-ориентированным.

2.4.1. Настройка основной формы

Новая форма имеет одинаковые имя (Name) и заголовок (Text) – Form1. С помощью мыши, можно задать необходимые размеры формы (захватив за кромку формы). Для задания свойств формы и элементов управления на форме используется окно свойств. Так, например, для изменения заголовка выберите форму, щелкнув кнопкой мыши по форме, и в окне свойств в

⁵ Например, событие, которое может произойти с программой или с операционной системой, под управлением которой она работает.

строке с названием **Text** наберите «Lab.#2. ст. гр. название группы. ФИО».

2.4.2. Размещение элементов управления на форме

Панель элементов содержит элементы управления, сгруппированные по типу. Основные элементы управления находятся в группе *Стандартные элементы управления* в Visual Studio (рис.2.10). Элементы управления перетаскиваются в нужное место формы путем захвата элемента левой кнопкой мышки. После размещения элемента управления на форме, его можно выделить и при этом получить доступ к его свойствам в окне свойств.

2.4.3. Размещение элемента ввода строки

Для ввода из формы в программу или вывода на форму информации, которая вмещается в одну строку, используют окно однострочного редактора текста, представляемого элементом управления **TextBox** (рис. 2.11).

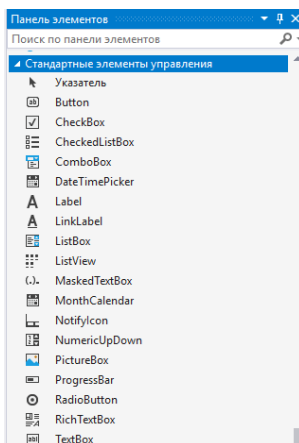


Рис. 2.10 – Панель стандартных элементов управления

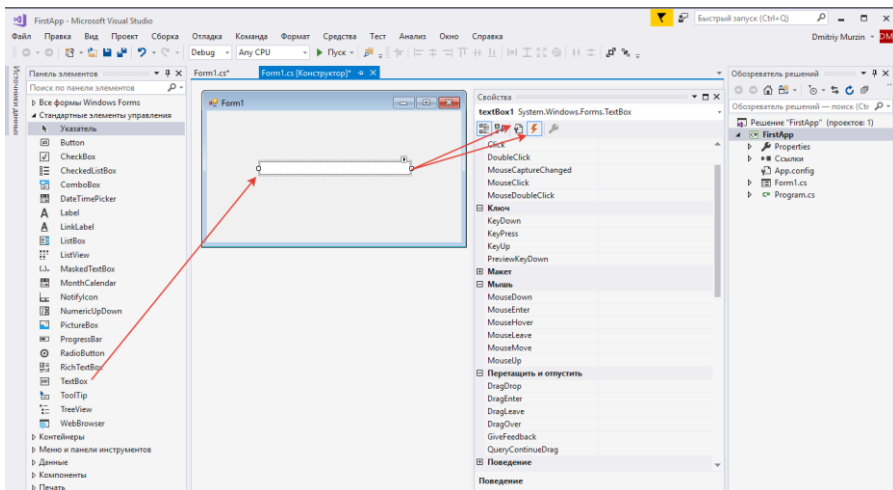


Рис. 2.11 – Работа с элементом управления TextBox

После помещения элемента в форму можно отрегулировать его размеры и положение. В тексте программы можно получить доступ к свойствам элемента (рис. 2.12), используя переменную **textBox1**, которая соответствует добавленному элементу управления. В этой переменной в свойстве **Text** будет содержаться строка символов (тип **string**) и отображаться в соответствующем окне **TextBox**. С помощью окна свойств можно установить необходимые шрифт и размер символов, отражаемых в строке **TextBox** (свойство **Font**) и другие настройки.

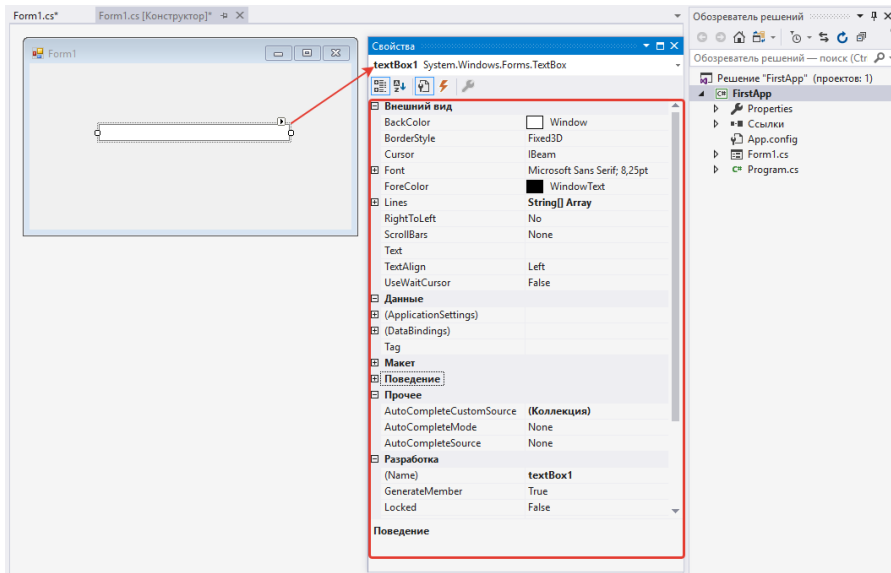
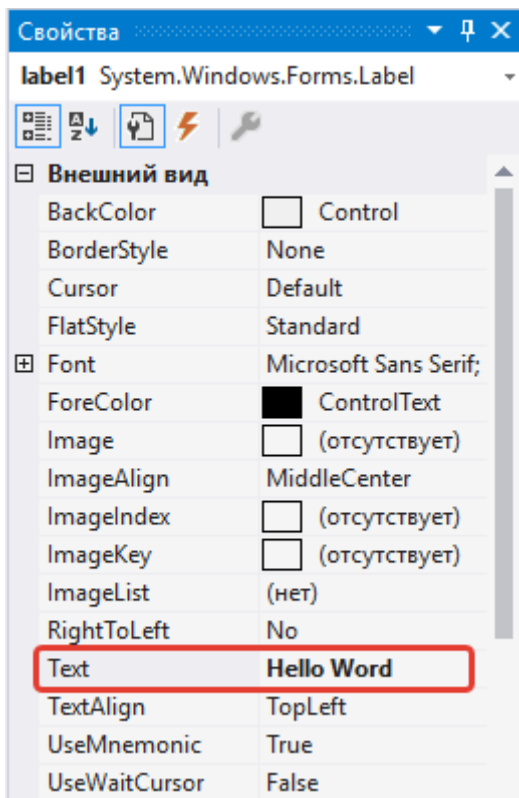


Рис. 2.12 – Свойства элемента управления TextBox

2.4.4 Метки

Для отображения простого текста на форме, доступного только для чтения, служит элемент **Label** (рис. 2.13). Чтобы задать отображаемый текст метки, надо установить свойство `Text` элемента.

Рисунок 2.13 — Свойства **Label**

2.4.5. Компонент класса **Button**

Компонент класса **Button** (рис. 2.14) предназначен для активизации выполнения требуемых действий. Часто используются следующие свойства (рис. 2.15) и события данного компонента:

- 1) **Text** – текст, который будет отображаться на кнопке;
- 2) **DialogResult** – используется, если кнопка расположена на диалоговом окне, и определяет способ закрытия формы;
- 3) **Click** – событие, возникающее при «нажатии» на кнопку, например при помощи мыши или клавиши **Enter**.

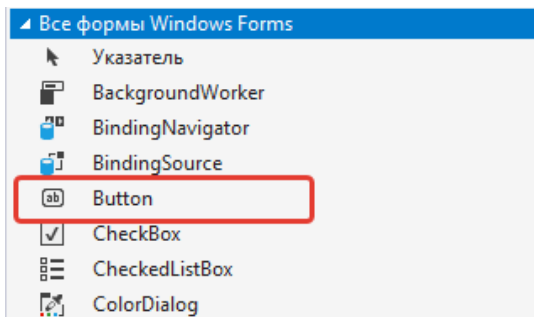


Рисунок 2.14 — Компонент Button

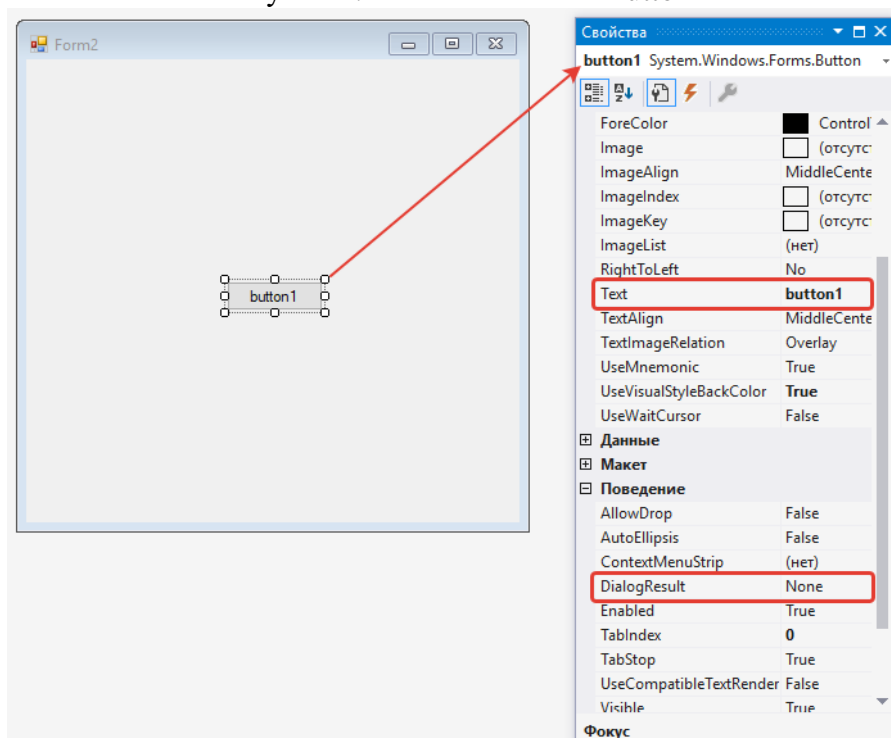


Рис. 2.15 – Свойства элемента управления Button

2.5 Введение в событийное программирование

Событийно-ориентированное программирование (СОП) – создание событийно-управляемых программ (event-driven programming). СОП – парадигма программирования, в которой выполнение программы определяется событиями, такими как, действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы.

Так, если обычный алгоритм – "последовательность команд, понятных исполнителю", то обработка событий описывается набором последовательностей команд, выполняемых при наступлении внешних по отношению к ней событий, время и последовательность наступления которых, как правило, заранее не известны и могут отличаться от запуска к запуску.

СОП можно также определить, как способ построения программы, где в коде выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события.

2.5.1. Написание программы обработки события

С элементами управления на форме и с самой формой могут происходить события во время работы программы. Например, на кнопку можно нажать и произойдет событие – нажатие кнопки. В окне с формой, может произойти такие события, как, создание окна, изменение размера окна и т. п. Такие события могут обрабатываться в программе с помощью обработчика события в виде специального метода.

2.5.2. Обработчик события нажатия кнопки

Если поместить на форму кнопку (элемент управления Button) и после этого два раза кликнуть мышью по кнопке, в программе появится код обработчика нажатия на кнопку:

```
private void button1_Click(object sender, EventArgs e){ }
```


Свой код добавляется между скобками { }, например, так:

```
private void button1_Click(object sender, EventArgs e){
    MessageBox.Show("Hello, " + textBox1.Text + "!");
}
```

2.5.3. Обработчик события загрузки формы


Для выбора соответствующего события выделенного элемента на форме используется окно свойств и его закладка  (рис. 2.16).



Рис. 2.16 – Иконка выбора событий

Если выбрать форму, и в окне событий установить событие Load, это приведет к тому, что в программе появится метод:

```
private void Form1_Load(object sender, EventArgs e){}
```

Между скобками { } вводится текст обработчика, например:

```
BackColor = Color.AntiqueWhite;
```

Каждый элемент управления имеет свой набор обработчиков событий. Если требуется удалить обработчик, то сначала нужно удалить строку с именем обработчика в окне свойств на закладке. Перечень наиболее часто применяемых событий:

- 1) Load – событие при загрузке формы.
- 2) KeyPress – событие при нажатии кнопки на клавиатуре⁶
- 3) KeyDown – событие при нажатии клавиши на клавиатуре⁷

⁶ параметр e.KeyChar имеет тип char и содержит код нажатой клавиши, например, клавиша Enter имеет код #13, клавиша Esc – #27 и т. д.

4) KeyUp – событие парное для KeyDown, возникает при отпускании ранее нажатой клавиши;

5) Click – событие при нажатии кнопки мыши в области элемента управления.

6) DoubleClick – событие при двойном нажатии кнопки мыши в области элемента управления.

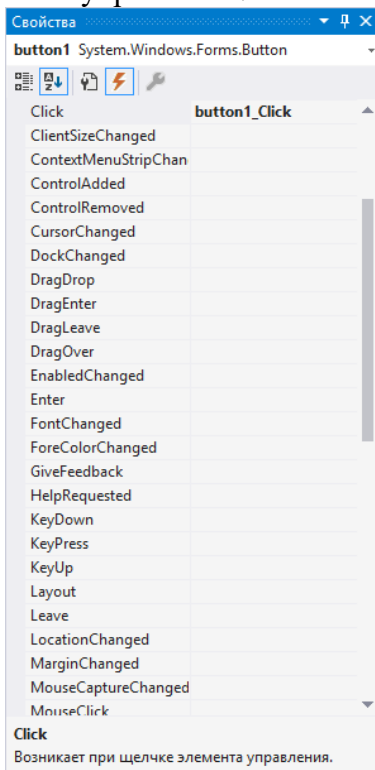


Рис. 2.17 – Панель событий компонента Button

⁷ обработчик этого события получает информацию в параметре e.KeyCode о нажатой клавише и состоянии клавиш Shift, Alt и Ctrl в параметре e.Modifiers, а также о нажатой кнопке мыши

2.5.4. Написание программы обработки события

Часто, для вывода сообщений применяется элемент **MessageBox**. Однако кроме собственно вывода строки сообщения данный элемент может устанавливать ряд настроек, которые определяют его поведение.

Для вывода сообщения в классе **MessageBox** предусмотрен метод **Show**, который имеет различные версии и может принимать ряд параметров. Рассмотрим одну из наиболее используемых версий:

```
public static DialogResult Show(
    string text,
    string caption,
    MessageBoxButtons buttons,
    MessageBoxIcon icon,
    MessageBoxDefaultButton defaultButton,
    MessageBoxOptions options
)
```

Здесь применяются следующие параметры:

text: текст сообщения

caption: текст заголовка окна сообщения

buttons: кнопки, используемые в окне сообщения.

Принимает одно из значений перечисления **MessageBoxButtons**:

- **AbortRetryIgnore**: три кнопки Abort (Отмена), Retry (Повтор), Ignore (Пропустить)
- **OK**: одна кнопка ОК
- **OKCancel**: две кнопки ОК и Cancel (Отмена)
- **RetryCancel**: две кнопки Retry (Повтор) и Cancel (Отмена)
- **YesNo**: две кнопки Yes и No
- **YesNoCancel**: три кнопки Yes, No и Cancel (Отмена)

Таким образом, в зависимости от выбора окно сообщения может иметь от одной до трех кнопок.

1) **icon**: значок окна сообщения. Может принимать одно из следующих значений перечисления **MessageBoxIcon**:

- **Asterisk, Information**: значок, состоящий из буквы *i* в нижнем регистре, помещенной в кружок
- **Error, Hand, Stop**: значок, состоящий из белого знака "X" на круге красного цвета.
- **Exclamation, Warning**: значок, состоящий из восклицательного знака в желтом треугольнике
- **Question**: значок, состоящий из вопросительного знака на периметре круга
- **None**: значок у сообщения отсутствует

2) **defaultButton**: кнопка, на которую по умолчанию устанавливается фокус. Принимает одно из значений перечисления **MessageBoxDefaultButton**:

- **Button1**: первая кнопка из тех, которые задаются перечислением **MessageBoxButtons**
- **Button2**: вторая кнопка
- **Button3**: третья кнопка

3) **options**: параметры окна сообщения. Принимает одно из значений перечисления **MessageBoxOptions**:

- **DefaultDesktopOnly**: окно сообщения отображается на активном рабочем столе.
- **RightAlign**: текст окна сообщения выравнивается по правому краю
- **RtlReading**: все элементы окна располагаются в обратном порядке справа налево
- **ServiceNotification**: окно сообщения отображается на активном рабочем столе, даже если в системе не зарегистрирован ни один пользователь

Нередко используется один параметр - текст сообщения (рис. 2.18). Но посмотрим, как использовать остальные парамет-

ры. Пусть у нас есть кнопка, в обработчике нажатия которой открывается следующее окно сообщения:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(
        "Выберите один из вариантов",
        "Сообщение",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Information,
        MessageBoxDefaultButton.Button1,
        MessageBoxOptions.DefaultDesktopOnly);
}
```

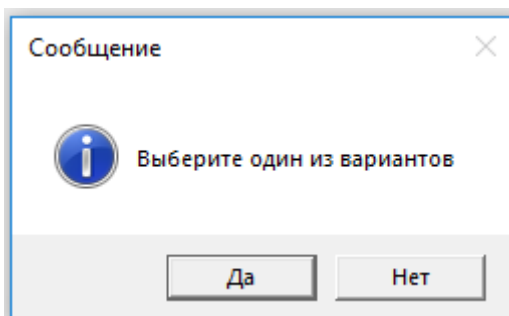


Рис. 2.18 — Внешний вид всплывающего окна

2.5.5. Оформление кнопки

Чтобы управлять внешним отображением кнопки, используют свойство **FlatStyle** со следующими значениями:

- **Flat** - Кнопка имеет плоский вид
- **Popup** - Кнопка приобретает объемный вид при наведении на нее указателя, в иных случаях она имеет плоский вид
- **Standard** - Кнопка имеет объемный вид (используется по умолчанию)
- **System** - Вид кнопки зависит от операционной системы

Как и для многих элементов управления, для кнопки можно задавать изображение с помощью свойства **BackgroundImage**.

Для управления размещением текста и изображения на кнопки используют свойство `TextImageRelation`:

- **Overlay**: текст накладывается на изображение
- **ImageAboveText**: изображение располагается над текстом
- **TextAboveImage**: текст располагается над изображением
- **ImageBeforeText**: изображение располагается перед текстом
- **TextBeforeImage**: текст располагается перед изображением

Например, установим для кнопки изображение. Для этого выберем кнопку и в окне Свойств нажмем на поле `Image` (не путать с `BackgroundImage`). Нам откроется диалоговое окно установки изображения показанное на рис. 2.19.

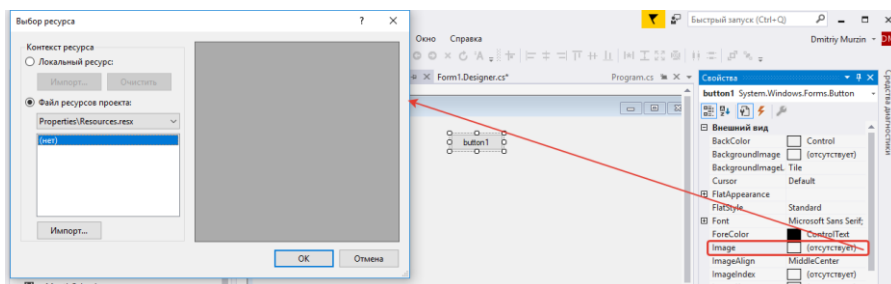


Рис. 2.19 — Диалоговое окно установки изображения

В этом окне выберем опцию `Local Resource` и нажмем на кнопку `Import`, после чего нам откроется диалоговое окно для выбора файла изображения.

После выбора изображения мы можем установить свойство **ImageAlign**, которое управляет позиционированием изображения на кнопке (рис. 2.20). Нам доступны 9 вариантов, с помощью которых мы можем прикрепить изображение к определен-

ной стороне кнопки. Например, значение по умолчанию — `MiddleCenter` означает позиционирование по центру.

В свойстве `TextImageRelation` можно установить `ImageBeforeText`. В итоге у кнопки сразу после изображения идет надпись (рис. 2.21).

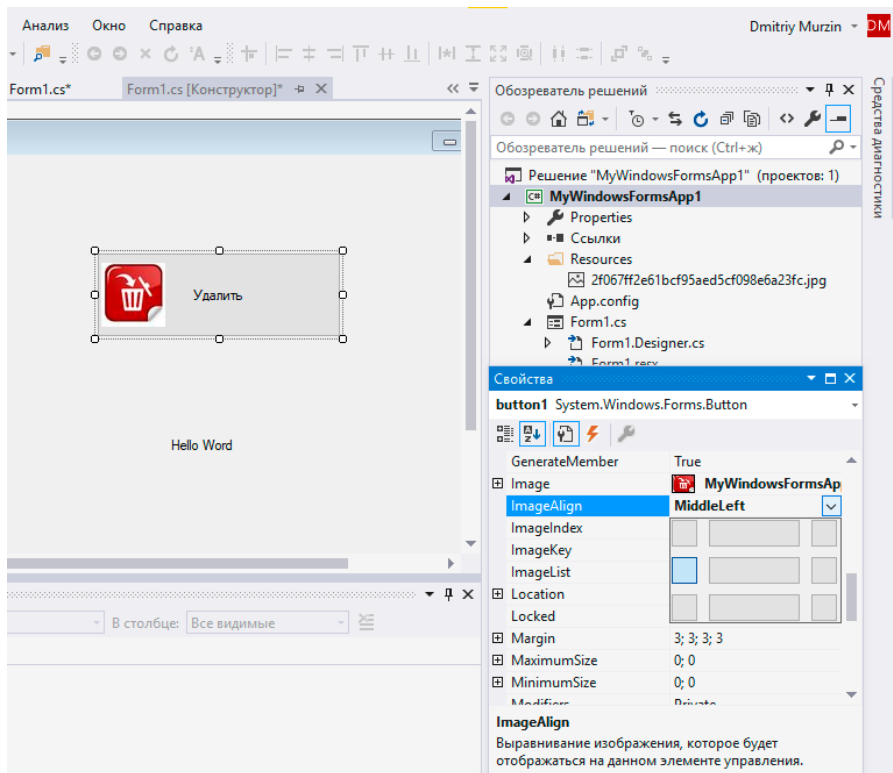


Рис. 2.20 — Окно управления позиционированием изображения на кнопке

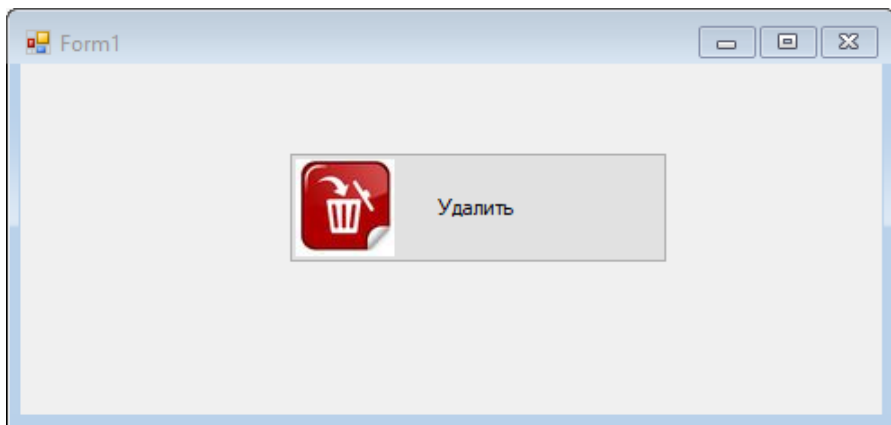


Рис. 2.21 — Итоговое изображение

2.5.6. Клавиши быстрого доступа

При работе с формами при использовании клавиатуры очень удобно пользоваться клавишами быстрого доступа. При нажатии на клавиатуре комбинации клавиш **Alt**+ некоторый символ, будет вызываться определенная кнопка. Например, зададим для некоторой кнопки свойство **Text** равное `&Аватар`. Первый знак - амперсанд - определяет ту букву, которая будет подчеркнута. В данном случае надпись будет выглядеть как Аватар. И теперь чтобы вызвать событие **Click**, нам достаточно нажать на комбинацию клавиш **Alt+A**.

2.5.7. Кнопки по умолчанию

Форма, на которой размещаются все элементы управления, имеет свойства, позволяющие назначать кнопку по умолчанию и кнопку отмены.

Так, свойство формы `AcceptButton` позволяет назначать кнопку по умолчанию, которая будет срабатывать по нажатию на клавишу **Enter**.

Аналогично работает свойство формы `CancelButton`, которое назначает кнопку отмены. Назначив такую кнопку, мы можем вызвать ее нажатие, нажав на клавишу **Esc**.

Индивидуальные задания

1) Разместите на форме четыре кнопки. Сделайте на кнопках следующие надписи, соответствующие 4 цветам. Создайте четыре обработчика события нажатия на данные кнопки, которые будут менять цвет формы в соответствии с текстом на кнопках.

2) Разместите на форме две кнопки и одну метку. Сделайте на кнопках следующие надписи: «*Hello*», «*Goodbye*». Создайте обработчики события нажатия на кнопки, которые будут менять текст метки на слова, написанные на кнопках. Создайте обработчик события создания формы, который будет устанавливать цвет формы и менять текст метки на строку «*Start*».


3) Разместите на форме две кнопки и одну метку. Сделайте на кнопках следующие надписи: «*Hide*», «*Show*». Создайте обработчики события нажатия на данные кнопки, которые будут скрывать или показывать метку. Создайте обработчик события создания формы (событие `Load`), который будет устанавливать цвет у формы и устанавливать текст метки «*Start*».

4) Разместите на форме три кнопки и одно поле ввода. Сделайте на кнопках следующие надписи: «*Hide*», «*Show*», «*Clear*». Создайте обработчики события нажатия на данные кнопки, которые будут скрывать или показывать поле ввода. При нажатии на кнопку «*Clear*» очистите текст из поля ввода.

5) Разместите на форме пять кнопок с цифрами от 1 до 5 и одну метку. При нажатии на кнопку значение кнопки выводится в метку. Восстановление исходного состояния происходит при нажатии на кнопку «*Start*».

6) Вывести простое информационное письмо с разрывом строки и с заголовком. «Задание 1. Здравствуйте! Пользователь».

7) Вывести информационное письмо совместно с информационным значком в окне сообщения. «Уважаемые водители — будьте внимательны на дороге!».

8) Вывести диалоговое окно с кнопками ОК (Основная), Отмена и со значком .

Порядок выполнения лабораторной работы

- 1) Запустите среду разработки Visual Studio.
- 2) Визуально спроектируйте форму основной программы в соответствии с индивидуальным заданием.
- 3) Выполните программирование события(ий) в программе.
- 4) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.
- 5) Выполните отчет по лабораторной работе.

Контрольные вопросы

1. Что понимается под термином «.NET Framework»?
2. Зависят ли приложения, разрабатываемые в .NET, от платформы?
3. Охарактеризуйте структуру среды разработки .NET Framework.
4. Что означает аббревиатура «CLR»?
5. Является ли среда CLR многоязычной?
6. Поясните процесс компиляции и выполнения программы в **.NET Framework**.
7. Перечислите и охарактеризуйте основные элементы, которые появляются при запуске интегрированной среды разработки **Visual Studio.NET**.

8. С помощью какого окна выбираются элементы управления для размещения их на форме?

9. Поясните процесс выполнения построения решения и отладки приложения.

10. Перечислите основные события для элементов формы и возможности их кодирования.

11. Какие значения могут принимать **MessageBoxButtons** ?

12. Какие значения могут принимать **MessageBoxIcon**?

13. Какие значения могут принимать **MessageBoxDefaultButton**?

14. Какие значения могут принимать **MessageBoxOptions**?

15. Каковы основные категории типов в языке C#?

16. Перечислите пять простых типов языка C#.

ПРАКТИЧЕСКАЯ РАБОТА № 3

Тема: «Знакомство с моделью программирования Windows Forms и базовыми компонентами ввода/вывода информации.»

Цель работы – научиться создавать Windows приложение с использованием элементов ввода-вывода в Visual Studio в соответствии с моделью программирования Windows Forms.

Теоретические сведения

3.1. Модель программирования Windows Forms

Для создания графического интерфейса пользователя (англ. graphical user interface, GUI) в среде Microsoft. NET применяется Windows Forms – стиль построения приложения на базе классов .NET Framework class library. Такие классы имеют собственную модель программирования, которая более совершеннее, чем модели, основанные на Win32 API, и они выполняются в управляемой среде .NET CLR.

Основное достоинство применения при программировании Windows Forms заключается в том, что Windows Forms предоставляет однородную программную модель.

В Windows Forms форма — это визуальная поверхность, на которой выводится информация для пользователя. Обычно приложение Windows Forms строится путем помещения элементов управления на форму и написания кода для реагирования на действия пользователя, такие как клики мыши или нажатия клавиш. Элемент управления — это отдельный элемент пользовательского интерфейса, предназначенный для отображения или ввода данных.

Для упрощения обозначений и организации сотен классов, .NET Framework class library представляет собой иерархическую структуру (рис. 3.1).

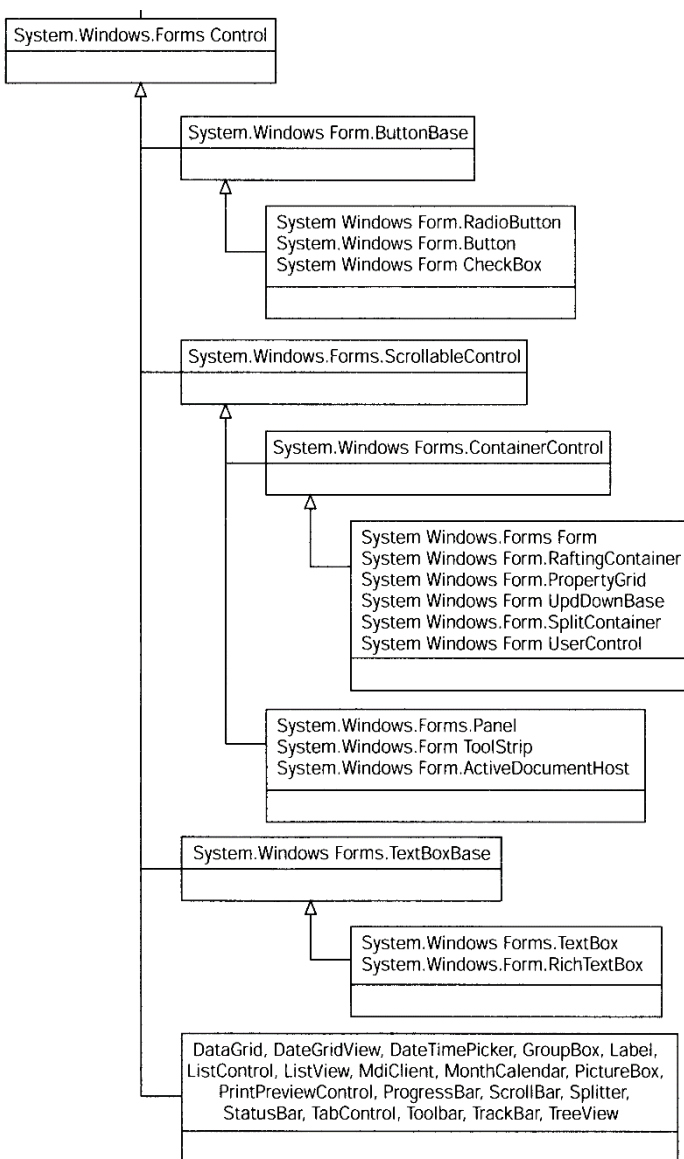


Рис. 3.1 – Иерархия компонентов Windows Forms

Приложения Windows Forms используют классы пространства имен System.Windows.Forms (табл. 3.1). Это пространство включает в себя такие классы, как Form, который моделирует поведение окон или форм; Menu, который представляет меню; Clipboard, который дает возможность приложениям Windows Forms использовать буфер обмена. Windows.Forms содержит также множество классов для построения диалога с пользователем, например, Button, TextBox, ListView, MonthCalendar и т.д.

Еще один важный блок приложения, который использует Windows Forms - класс Application. Данный класс содержит статический метод Run, который загружает приложение и отображает окно. Модель программирования Windows Forms использует механизм обработки событий в ответ на ввод/вывод данных средств управления и других элементов GUI приложения. В C# и других языках, которые поддерживают CLR, события - члены типа класса наравне с методами, полями и свойствами. Фактически все управляющие классы (control classes) Windows Forms создают события. Например, кнопка button после нажатия создает событие Click. Форма, которая должна ответить на нажатие кнопки может использовать следующий код, чтобы соединить кнопку с обработчиком события Click:

```
MyButton.Click += new EventHandler (OnButtonClicked);
...
private void OnButtonClicked (object sender, EventArgs
e) {
    MessageBox.Show ("Click!");
}
```

EventHandler - специальный обработчик событий, который выполняет метод OnButtonClicked когда MyButton создает событие Click. Первый параметр OnButtonClicked идентифицирует объект, который вызвал событие. Второй параметр в основном бессмысленен для события Click, но используется некоторыми другими типами событий, чтобы передать дополнительную информацию.

Таблица 3.1 – Пространство имен System.Windows.Forms.

| Класс | Назначение |
|--|--|
| Application | Этот класс представляет саму суть приложения Windows Forms. При помощи методов этого класса вы можете обрабатывать сообщения Windows, запускать и прекращать работу приложения и т. п. |
| ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox | Эти классы (а также многие аналогичные им) представляют элементы графического интерфейса |
| Form | Этот тип представляет главную форму (диалоговое окно) приложения Windows Forms |
| ColorDialog, FileDialog, FontDialog, PrintPreviewDialog | В .NET предусмотрено множество готовых к употреблению диалоговых окон для выбора цветов, файлов, шрифтов и т. п. |
| Menu, MainMenu, MenuItem, ContextMenu | Эти типы предназначены для создания ниспадающих и контекстных меню |
| Clipboard, Help, Timer, Screen, ToolTip, Cursors | Разнообразные вспомогательные типы для организации интерактивных графических интерфейсов |
| StatusBar, Splitter, ToolBar, ScrollBar | Дополнительные элементы управления, размещаемые на форме |

3.2 Понятие пространств имен

При написании кода современных программ возникают следующие проблемы:

1. в больших программах сложнее разбираться и соответственно их сложнее сопровождать;
2. чем больше кода, тем больше классов, в которых больше методов, что требует отслеживания все большего количества имен.

По мере роста количества имен растет и вероятность сбоя при сборке проекта из-за конфликта двух и более имен⁸. Данная проблема решается с помощью пространств имен (namespaces), т.е. путем создания контейнеров для таких элементов, как классы⁹. Используя ключевое слово namespace, можно внутри пространства имен под названием MyTestHello создать класс по имени Hello:

```
namespace MyTestHello {
    class Hello {
        ...
    }
}
```

При использовании такого кода в программе можно обращаться к классу Hello с помощью ссылки MyTestHello.Hello. Если другой разработчик также создаст класс Hello, но уже в другом пространстве имен, например, NewNamespace, и после установления сборки, содержащую этот класс, на компьютер, программы не изменят своего поведения, поскольку в них используется класс MyTestHello.Hello. Если понадобится обратиться к

⁸ Ситуация еще больше усложняется, когда программа ссылается на сборки, написанные другими разработчиками, которые также использовали различные имена.

⁹ Два класса с одинаковыми именами не приведут к возникновению путаницы, если они находятся в разных пространствах имен.

классу Hello другого разработчика, нужно будет указать на него как на `NewNamespace.Hello`.

Определение всех классов в пространствах имен является рекомендуемой нормой, и среда Visual Studio 2017 следует этой рекомендации, используя в качестве пространства имен верхнего уровня название вашего проекта. В библиотеке классов .NET Framework также соблюдается эта норма, поэтому каждый класс в этой библиотеке находится в своем пространстве имен. Чтобы не указывать каждый раз полное имя класса используется директива `using`, с помощью которой пространство имен вводится в область видимости:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

Подробнее про пространство имен можно ознакомиться в разделе документации, перейдя по ссылке <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/namespaces/>

3.3 Класс `System.Windows.Forms.Application`

Класс `Application` предоставляет группу статических методов и свойств для управления процессом запуска и остановки приложения, а также обеспечивает доступ к сообщениям Windows, обрабатываемым приложением. Наиболее важные методы этого класса перечислены в таблице 3.2.

Класс `Application` определяет множество статических свойств (таблица 3.3), большинство из которых доступны только для чтения. Многие из этих свойств предназначены для получения общей информации о приложении, такой как название компании, номер версии и т.п.

Таблица 3.2 – Наиболее важные методы типа Application.

| Метод класса | Назначение Application |
|--------------------|--|
| AddMessageFilter() | Эти методы позволяют приложению перехватывать сообщения RemoveMessageFilter() и выполнять с этими сообщениями необходимые предварительные действия ¹⁰ . |
| DoEvents() | Позволяет обработать сообщения в очереди. |
| Exit() | Завершает работу приложения |
| ExitThred() | Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток |
| OLERequired() | Инициализирует библиотеки OLE. |
| Run() | Запускает стандартный цикл работы с сообщениями для текущего потока |

Таблица 3.2 – Наиболее важные свойства типа Application.

| Свойство | Назначение |
|-----------------------|--|
| CommonAppDataRegistry | Путь к данным, общий для всех пользователей приложения. |
| CompanyName | Возвращает имя компании |
| CurrentCulture | Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток |
| CurrentInputLanguage | Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком |
| ProductName | Для получения имени программного продукта, которое ассоциировано с данным приложением |
| ProductVersion | Позволяет получить номер версии программного продукта |
| ExecutablePath | Путь и имя исполняемого файла, запускающего приложение. |
| StartupPath | Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в |

¹⁰ Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter

3.4 Класс `System.Windows.Forms.Control`

Класс `System.Windows.Forms.Control` задает общее поведение, ожидаемое от любого GUI-типа. Базовые члены `Control` позволяют указать размер и позицию элемента управления, выполнить захват событий клавиатуры и мыши, получить и установить фокус ввода, задать и изменить видимость членов и т.д.

3.4.1 Размер и местоположение

Размер и местоположение элементов управления определяются свойствами `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right`, вместе с дополняющими их `Size` и `Location`. Отличие состоит в том, что `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right` принимают одно целое значение. `Size` принимает значение структуры `Size`, а `Location` — значение структуры `Point`. Структуры `Size` и `Point` включают в себя координаты `X`, `Y`. `Point` обычно описывает местоположение, а `Size` — высоту и ширину объекта. `Size` и `Point` определены в пространстве имен `System.Drawing`.

Свойство `Bounds` возвращает объект `Rectangle`, представляющий экранную область, занятую элементом управления. Эта область включает полосы прокрутки и заголовка. `Rectangle` также относится к пространству имен `System.Drawing`. Свойство `ClientSize` — структура `Size`, представляющая клиентскую область элемента управления за вычетом полос прокрутки и заголовка.

Методы `PointToClient` и `PointToScreen` — методы преобразования, которые принимают `Point` и возвращают `Point`. Метод `PointToClient`¹¹ принимает структуру `Point`, представляющую экранные координаты, и транслирует их в коор-

¹¹ Это удобно для операций перетаскивания.

динаты текущего клиентского объекта. Метод `PointToScreen` выполняет обратную операцию — принимает координаты в клиентском объекте и транслирует их в экранные координаты.

Методы `RectangleToScreen` и `ScreenToRectangle` выполняют те же операции, но со структурами `Rectangle` вместо `Point`.

Свойство `Dock` определяет, к какой грани родительского элемента управления должен пристыковываться данный элемент. Перечисление `DockStyle` задает возможные значения этого свойства. Они могут быть такими: `Top`, `Bottom`, `Right`, `Left`, `Fill` и `None`. Значение `Fill` устанавливает размер данного элемента управления равным размеру родительского.

Свойство `Anchor` (якорь) прикрепляет грань данного элемента управления к грани родительского элемента управления. Это отличается от стыковки (`docking`) тем, что не устанавливает грань дочернего элемента управления точно на грань родительского, а просто выдерживает постоянное расстояние между ними. Свойство `Anchor` принимает значения из перечисления `AnchorStyle`, а именно: `Top`, `Bottom`, `Left`, `Right` и `None`. Устанавливая эти значения, можно заставить элемент управления изменять свой размер динамически вместе с родителем. Таким образом, кнопки и текстовые поля не будут усечены или скрыты при изменении размеров формы пользователем. Свойства `Dock` и `Anchor` применяются в сочетании с компоновками элементов управления `Flow` и `Table` (о которых мы поговорим позднее в этой главе) и позволяют создавать очень сложные пользовательские окна.

3.4.2 Внешний вид

Свойства, имеющие отношение к внешнему виду элемента управления — это `BackColor` и `ForeColor`, которые

принимают объект `System.Drawing.Color` в качестве значения. Свойство `BackColorImage` принимает объект графического образа как значение. Класс `System.Drawing.Image` — абстрактный класс, служащий в качестве базового для классов `Bitmap` и `Metafile`. Свойство `BackColorImageLayout` использует перечисление `ImageLayout` для определения способа отображения графического образа в элементе управления: `Center`, `Tile`, `Stretch`, `Zoom` или `None`.

Свойства `Font` и `Text` работают с надписями. Чтобы изменить `Font`, необходимо создать объект `Font`. При создании этого объекта указывается имя, стиль и размер шрифта.

3.4.3 Взаимодействие с пользователем

Взаимодействие с пользователем лучше всего описывается серией событий, которые генерирует элемент управления и на которые он реагирует. Некоторые из наиболее часто используемых событий: `Click`, `DoubleClick`, `KeyDown`, `KeyPress`, `Validating` и `Paint`.

События, связанные с мышью — `Click`, `DoubleClick`, `MouseDown`, `MouseUp`, `MouseEnter`, `MouseLeave` и `MouseHover` — описывают взаимодействие мыши и экранного элемента управления. События `Click`, и `DoubleClick` принимают в качестве аргумента `EventArgs`, а события `MouseDown` и `MouseUp` принимают `MouseEventArgs`.

Для простейших случаев обработки событий клавиатуры можно использовать событие `KeyPress`, которое принимает `KeyPressEventArgs`. Эта структура включает `KeyChar`, представляющий символ нажатой клавиши. Свойство `Handled` используется для определения того, было ли событие обработано. Установив значение `Handled` в `true`, можно добиться то-

го, что событие не будет передано операционной системе для совершения стандартной обработки.

В случае, если необходима дополнительная информация о нажатой клавише, используют события `KeyDown` или `KeyUp`. Оба принимают структуру `KeyEventArgs`. Свойства `KeyEventArgs` включают признак одновременного состояния клавиш `<Ctrl>`, `<Alt>` или `<Shift>`. Свойство `KeyCode` возвращает значение типа перечисления `Keys`, идентифицирующее нажатую клавишу. В отличие от свойства `KeyPressEventArgs.KeyChar`, свойство `KeyCode` сообщает о каждой клавише клавиатуры, а не только о буквенно-цифровых клавишах. Свойство `KeyData` возвращает значение типа `Keys`, а также устанавливает модификатор.

Значение модификатора сопровождается значением клавиши, объединяясь с ним двоичной логической операцией “ИЛИ”. Таким образом, можно получить информацию о том, была ли одновременно нажата клавиша `<Shift>` или `<Ctrl>`. Свойство `KeyValue` — целое значение из перечисления `Keys`. Свойство `Modifiers` содержит значение типа `Keys`, представляющее нажатые модифицирующие клавиши. Если было нажато более одной такой клавиши, их значения объединяются операцией “ИЛИ”. События клавиш поступают в следующем порядке:

1. `KeyDown`
2. `KeyPress`
3. `KeyUp`

События `Validating`, `Validated`, `Enter`, `Leave`, `GotFocus` и `LostFocus` имеют отношение к получению фокуса¹² элементами управления и утере его. Это случается, когда пользователь нажатием клавиши `<Tab>` переходит к данному элементу управления либо выбирает его мышью. События

¹² т.е. когда становятся активными

`Validating` и `Validated` возбуждаются при проверке данных в элементе управления. Эти события принимают аргумент `CancelEventArgs`. С его помощью можно отменить последующие события, установив свойство `Cancel` в `true`. Если разрабатывается собственный проверочный код, и проверка завершается неудачно, то в этом случае можно установить `Cancel` в `true` — тогда элемент управления не утратит фокус. `Validating` происходит во время проверки, а `Validated` — после нее. Порядок возникновения событий следующий:

1. `Enter`
2. `GotFocus`
3. `Leave`
4. `Validating`
5. `Validated`
6. `LostFocus`

Понимание последовательности этих событий важно, чтобы избежать рекурсивных ситуаций.

3.4.4 Функциональность Windows

Пространство имен `System.Windows.Forms` — одно из немногих, полагающихся на функциональность операционной системы Windows.

Функциональность, которая поддерживает взаимодействие с Windows, включает свойства `Handle` и `IsHandleCreated`. Свойство `Handle` возвращает `IntPtr`, содержащий `HWND` (дескриптор окна) элемента управления. Дескриптор окна — это `HWND`, уникально идентифицирующий окно. Элемент управления может рассматриваться как окно, поэтому у него есть соответствующий `HWND`. Свойство `Handle` можно использовать для обращения к любым вызовам Win32 API.

Для получения доступа к внутренним сообщениям Windows можно переопределить метод `WndProc`. Метод

WndProc принимает в качестве параметра объект Message. Этот объект представляет собой просто оболочку для сообщения окна. Он содержит свойства HWND, LParam, WParam, Msg и Result. Если нужно, чтобы сообщение было обработано системой, его потребуется передать на обработку базовому методу `base.WndProc(msg)`. Если же нужно обработать его в вашем приложении специальным образом, то передавать его этому методу не следует.

3.5 Основные свойства форм

С помощью специального окна Properties (Свойства) Visual Studio предоставляет удобный графический интерфейс для управления свойствами элемента (рис. 3.2):

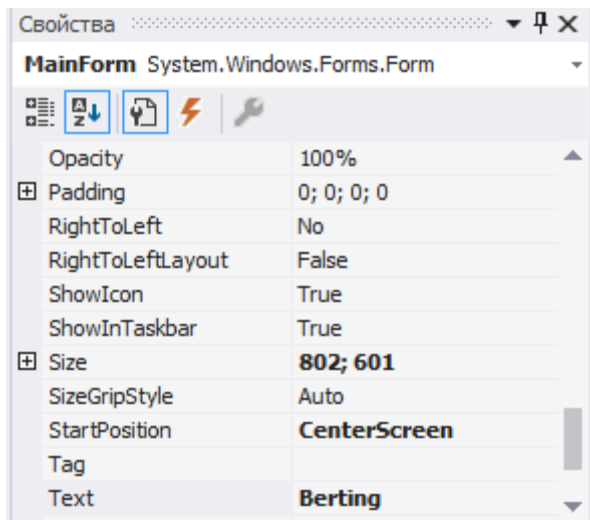


Рисунок 3.2 – Панель свойства Visual Studio

Большинство этих свойств оказывает влияние на визуальное отображение формы.

Рассмотрим основные свойства:

- **Name**: устанавливает имя формы - точнее имя класса, который наследуется от класса **Form**;
- **BackColor**: указывает на фоновый цвет формы. «Щелкнув» на это свойство, можно выбрать тот цвет, который нам подходит из списка предложенных цветов или цветовой палитры;
- **BackgroundImage**: указывает на фоновое изображение формы;
- **BackgroundImageLayout**: определяет, как изображение, заданное в свойстве **BackgroundImage**, будет располагаться на форме;
- **ControlBox**: указывает, отображается ли меню формы. В данном случае под меню понимается меню самого верхнего уровня, где находятся «иконка» приложения, заголовок формы, а также кнопки минимизации формы и «крестик». Если данное свойство имеет значение **false**, то на форме не будем ни иконки, ни крестика (с помощью которого обычно закрывается форма);
- **Cursor**: определяет тип курсора, который используется на форме;
- **Enabled**: если данное свойство имеет значение **false**, то форма не сможет получать «ввод от пользователя», то есть пользователь не сможет нажать на кнопки формы, ввести текст в текстовые поля и т.д.;
- **Font**: задает шрифт для всей формы и всех помещенных на нее элементов управления. Однако, задав у элементов формы свой шрифт, можно переопределить его;
- **ForeColor**: цвет шрифта на форме;
- **FormBorderStyle**: указывает, как будет отображаться граница формы и строка заголовка. Устанавливая данное свойство в **None** можно создавать внешний вид приложения произвольной формы;

- `HelpButton`: указывает, отображается ли кнопка справки формы;
- `Icon`: задает иконку формы;
- `Location`: определяет положение по отношению к верхнему левому углу экрана, если для свойства `StartPosition` установлено значение `Manual`;
- `MaximizeBox`: указывает, будет ли доступна кнопка максимизации окна в заголовке формы;
- `MinimizeBox`: указывает, будет ли доступна кнопка минимизации окна формы;
- `MaximumSize`: задает максимальный размер формы;
- `MinimumSize`: задает минимальный размер формы;
- `Opacity`: задает прозрачность формы;
- `Size`: определяет начальный размер формы;
- `StartPosition`: указывает на начальную позицию, с которой форма появляется на экране;
- `Text`: определяет заголовок формы;
- `TopMost`: если данное свойство имеет значение `true`, то форма всегда будет находиться поверх других окон;
- `Visible`: видна ли форма, если необходимо скрыть форму от пользователя, то необходимо задать данному свойству значение `false`;
- `WindowState`: указывает, в каком состоянии форма будет находиться при запуске: в нормальном, максимизированном или минимизированном.

3.5.1 Установка размеров формы

Для установки размеров формы можно использовать такие свойства как `Width/Height` или `Size`. `Width/Height` принимают числовые значения. При установке размеров через свойство `Size`, надо присвоить свойству объект типа `Size`:

```
this.Size = new Size(200,150);
```

Объект `Size`, в свою очередь, принимает в конструкторе числовые значения для установки ширины и высоты.

3.5.2 Начальное расположение формы

Начальное расположение формы устанавливается с помощью свойства `StartPosition`, которое может принимать одно из следующих значений:

- `Manual`: положение формы определяется свойством `Location`;
- `CenterScreen`: Положение формы в центре экрана;
- `WindowsDefaultLocation`: позиция формы на экране задается системой `Windows`, а размер определяется свойством `Size`;
- `WindowsDefaultBounds`: начальная позиция и размер формы на экране задается системой `Windows`;
- `CenterParent`: положение формы устанавливается в центре родительского окна;

Все эти значения содержатся в перечислении `FormStartPosition`, поэтому, чтобы, например, установить форму в центре экрана, нам надо записать так:

```
this.StartPosition = FormStartPosition.CenterScreen;
```

3.5.3 Фон и цвета формы

Чтобы установить цвет как фона формы, так и шрифта, надо использовать цветовое значение, хранящееся в структуре `Color`:

```
this.BackColor = Color.Aquamarine;
```

```
this.ForeColor = Color.Red;
```

Кроме того, можно в качестве фона задать изображение в свойстве `BackgroundImage`, выбрав его в окне свойств или в коде, указав путь к изображению:

```
this.BackgroundImage =
```

```
Image.FromFile("C:\\Users\\Student\\Pictures\\1.jpg");
```

Чтобы настроить нужное отображение фоновой картинки, надо использовать свойство `BackgroundImageLayout`, которое может принимать одно из следующих значений:

- `None`: Изображение помещается в верхнем левом углу формы и сохраняет свои первоначальные значения;
- `Tile`: Мозаичное заполнение изображением;
- `Center`: Изображение располагается по центру формы;
- `Stretch`: Изображение растягивается до размеров формы без сохранения пропорций;
- `Zoom`: Изображение растягивается до размеров формы с сохранением пропорций;

Например, расположим форму по центру экрана:

```
this.StartPosition = FormStartPosition.CenterScreen;
```

3.6 Базовые компоненты ввода/вывода информации.

3.6.1 Метки (Label)

Метки `Label` (рис. 3.3) применяются для представления пользователю описательного текста. Текст может иметь отношение к другому элементу управления либо к текущему состоянию системы. Обычно метки помещаются рядом с текстовыми полями. Метка предлагает пользователю описание типа данных для ввода в текстовое поле.

Элемент управления `Label` всегда доступен только для чтения — пользователь не может изменить значение строки в его свойстве `Text` (рис. 3.4). Однако возможно программное изменение значения свойства `Text`.

Свойство `UseMnemonic` позволяет включить функциональность клавиши доступа. Когда букве в свойстве `Text` предшествует символ амперсанда (&), эта буква высвечивается с подчеркиванием. Нажатие клавиши <Alt> в сочетании с клавишей этой буквы устанавливает фокус на следующий (в порядке

обхода) после метки элемент управления. Если свойство Text уже содержит в тексте амперсанд, то добавление второго не вызовет подчеркивания буквы. Например, если текстом метки должно быть Copy & Paste, то свойство должно иметь значение Copy && Paste.

```
label1.UseMnemonic = true;  
label1.Text = "&Print";  
label2.UseMnemonic = true;  
label2.Text = "&Copy && Paste";
```

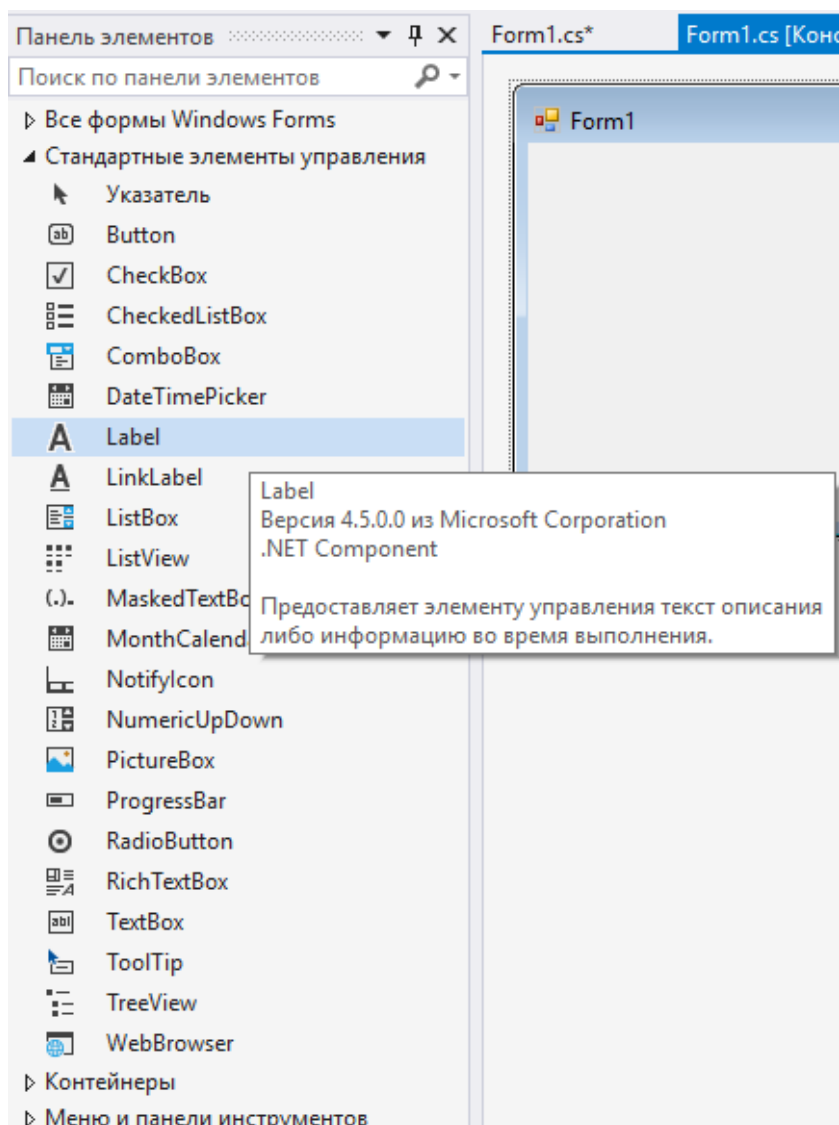


Рисунок 3.3 – Элемент управления Label

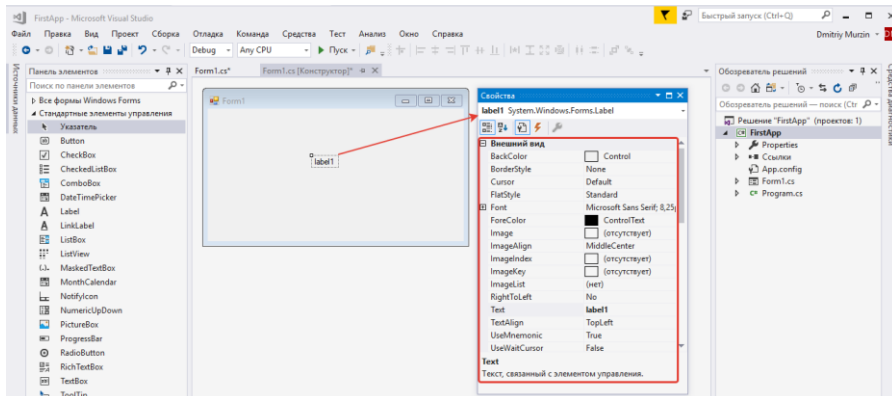


Рисунок 3.4 – Свойства элемента управления Label

Свойство `AutoSize` содержит булевское значение, указывающее на то, что `Label` может автоматически изменять свой размер в соответствии со размером текста. Перечень основных свойств `Label` показан в таблице 3.3.

Таблица 3.3 – Наиболее важные свойства `Label`.

| Свойство | Описание |
|----------|--|
| Name | Имя компонента. Используется в программе для доступа к свойствам компонента |
| Text | Отображаемый текст |
| Location | Положение компонента на поверхности формы |
| AutoSize | Признак автоматического изменения размера компонента. Если значение равно <code>True</code> , то при изменении значения свойства <code>Text</code> (или <code>Font</code>) автоматически изменяется размер компонента |

| | |
|------|---|
| Size | Размер компонента. Определяет (если значение AutoSize равно False) размер компонента. |
|------|---|

Продолжение таблицы Таблица 3.3

| Свойство | Описание |
|-------------|---|
| MaximumSize | Если значение свойства AutoSize равно True, то задает максимально допустимый размер компонента. Свойство MaximumSize.Width задает максимально возможную ширину области, свойство MaximumSize.Height — высоту. |
| Font | Шрифт, используемый для отображения текста |
| ForeColor | Цвет текста, отображаемого в поле компонента |
| BackColor | Цвет закрашки области вывода текста |
| TextAlign | Способ выравнивания текста в поле компонента. |
| BorderStyle | Вид рамки (границы) компонента. По умолчанию граница вокруг поля Label отсутствует (значение свойства равно None). Граница компонента может быть обычной (Fixed3D) или тонкой (FixedSingle) |

3.6.2 LinkLabel

Тип меток класса LinkLabel предназначен для вывода ссылок, аналогичный веб-ссылкам.

Так же, как и для обычных ссылок веб-страниц, можно определить следующие свойства цвета ссылки:

- `ActiveLinkColor` задает цвет ссылки при нажатии;
- `LinkColor` задает цвет ссылки до нажатия, по которой еще не было переходов;

- `VisitedLinkColor` задает цвет ссылки, по которой уже были переходы;

Свойство `LinkBehavior` управляет поведением ссылки:

- `SystemDefault`: для ссылки устанавливаются системные настройки;
- `AlwaysUnderline`: ссылка всегда подчеркивается;
- `HoverUnderline`: ссылка подчеркивается только при наведении на нее курсора мыши;
- `NeverUnderline`: ссылка никогда не подчеркивается;

По умолчанию весь текст на данном элементе считается ссылкой. Однако с помощью свойства `LinkArea` можно изменить область ссылки (рис. 3.5).

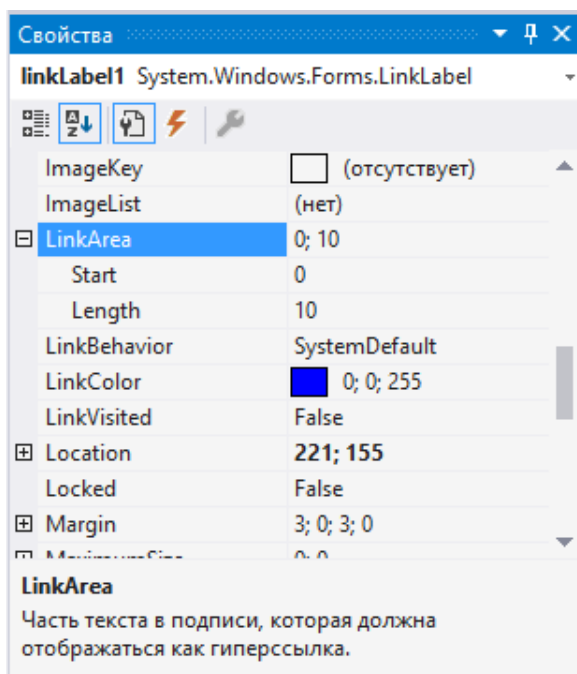


Рисунок 3.5 – Свойство элемента LinkLabel: LinkArea

Чтобы выполнить переход по ссылке «по нажатию на нее», надо дополнительно написать программный код. Данный код должен обрабатывать событие `LinkClicked`, которое есть у элемента `LinkLabel`.

Например, пусть на форме есть элемент ссылки (рис. 3.6), называемый `linkLabel1`:

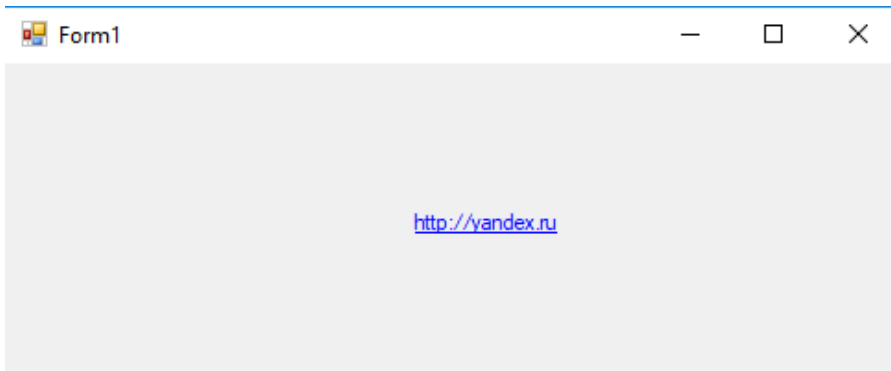


Рисунок 3.6 – Пример применения элемента ссылки

Чтобы перейти по ссылке, необходимо задать обработчик `LinkClicked`:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        // задаем обработчик события
        linkLabel1.LinkClicked += linkLabel1_LinkClicked;
    }
    private void linkLabel1_LinkClicked(object sender,
        LinkLabelLinkClickedEventArgs e) {
        Sys-
        tem.Diagnostics.Process.Start(@"http://yandex.ru");
    }
}
```

Метод `System.Diagnostics.Process.Start()` «откроет» данную ссылку в веб-браузере, который установлен в системе браузером по умолчанию.

3.6.3 Текстовое поле `TextBox`

Использование компонента класса `TextBox` является простейшим способом организации ввода/вывода данных. Компонент расположен в разделе компонентов «Стандартные элементы управления» (рис. 3.7).

`TextBox`, `RichTextBox` и `MaskedTextBox` унаследованы от `TextBoxBase`. Класс `TextBoxBase` представляет такие свойства, как `MultiLine` и `Lines`. Свойство `MultiLine` — булевское значение, позволяющее элементу управления `TextBox` отображать текст в более чем одной строке. При этом каждая строка в текстовом окне является частью массива строк. Этот массив доступен через свойство `Lines`. Свойство `Text` возвращает полное содержимое текстового окна в виде одной строки. `TextLength` — общая длина текста. Свойство `MaxLength` ограничивает длину текста определенной величиной.

`SelectedText`, `SelectionLength` и `SelectionStart` имеют дело с текущим выделенным текстом в текстовом окне. Выделенный текст подсвечивается, когда элемент управления получает фокус.

`TextBox` добавляет множество интересных свойств. `AcceptsReturn` — булевское значение, позволяющее `TextBox` воспринимать клавишу `<Enter>` как символ новой строки либо активизировать кнопку по умолчанию на форме. Когда это свойство имеет значение `true`, то нажатие `<Enter>` создает новую строку в `TextBox`.

Свойство `CharacterCasing` определяет регистр текста в текстовом окне. Перечисление `CharacterCasing` со-

держит три значения: Lower, Normal и Upper. Значение Lower переводит в нижний регистр весь текст, независимо от того, как он был введен, Upper переводит весь текст в верхний регистр, а Normal отображает текст так, как он был введен.

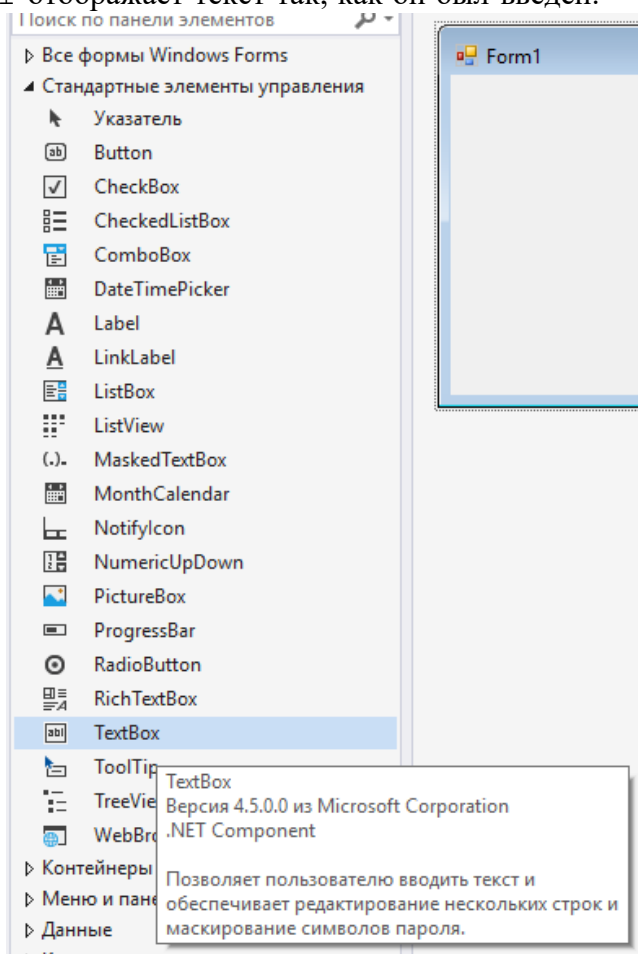


Рисунок 3.7 – Элемент управления TextBox

Свойство `PasswordChar` позволяет указать символ, который будет отображаться при вводе пользователем всех символов в текстовом окне. Это применяется при вводе паролей и PIN-кодов. Свойство `text` вернет действительный введенный текст; свойство `PasswordChar` касается только отображения символов.

`RichTextBox` — элемент управления, служащий для редактирования текста с расширенными возможностями форматирования. Как следует из его названия, `RichTextBox` использует `Rich Text Format (RTF)` для обработки специального форматирования.

Изменения формата обеспечиваются свойствами `SelectionFont`, `SelectionColor` и `SelectionBullet`, а форматирование параграфов — свойствами `SelectionIndent`, `SelectionRightIndent` и `SelectionHangingIndent`. Все свойства их группы `Selection` работают одинаково. Если выделена часть текста, то изменение свойства касается этого выделенного фрагмента. Если же выделенного фрагмента нет, то изменения затрагивают любой текст, вставляемый справа от текущей позиции вставки.

Текст данного элемента управления может быть извлечен из свойства `Text` либо `Rtf`. Свойство `Text` возвращает простой текст элемента управления, в то время как `Rtf` — форматированный текст.

Метод `LoadFile` может загружать текст из файла двумя различными способами. Он может использовать либо строку, представляющую путь к файлу, либо потоковый объект.

3.6.4 Авто заполнение текстового поля

Элемент `TextBox` позволяет создавать автозаполняемые поля. Для этого достаточно привязать свойство

AutoCompleteCustomSource элемента TextBox к некоторой коллекции, из которой берутся данные для заполнения поля.

Пусть на форму размещено текстовое поле и описан код события загрузки следующие строки:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        AutoCompleteStringCollection source =
new AutoCompleteStringCollection() {
    "Кузнецов",
    "Иванов",
    "Петров",
    "Кустов"
};
    textBox1.AutoCompleteCustomSource =
source;

    textBox1.AutoCompleteMode =
    AutoCompleteMode.SuggestAppend;
    textBox1.AutoCompleteSource =
    AutoCompleteSource.CustomSource;
    }
}
```

Режим автодополнения, представленный свойством AutoCompleteMode, может принимать следующие значения:

- None: отсутствие автодополнения;
- Suggest: предлагает варианты для ввода, но не дополняет;
- Append: дополняет введенное значение до строки из списка, но не предлагает варианты для выбора;
- SuggestAppend: одновременно и предлагает варианты для автодополнения, и дополняет введенное пользователем значение;

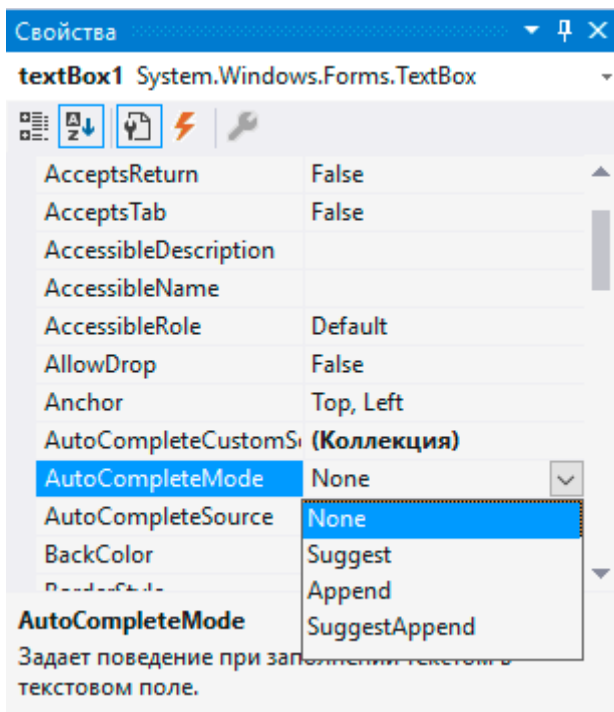


Рисунок 3.8 – Опции настройки AutoCompleteMode

3.6.5 Перенос по словам

Чтобы текст в элементе TextBox «переносился» по словам, надо установить свойство WordWrap равным true. То есть, если очередное слово не умещается в текущей строке, то оно переносится на следующую строку. Данное свойство будет работать только для многострочных текстовых полей.

3.6.6 Ввод пароля

Свойства PasswordChar и UseSystemPasswordChar позволяют сделать текстовое поле полем для ввода пароля.

Свойство `PasswordChar` не имеет значение «по умолчанию». Если установить его «значением» каким-нибудь символом, то этот символ будут отображаться при вводе любых символов в «парольное» текстовое поле.

Свойство `UseSystemPasswordChar` активизирует похожее действие. Если установить его значение в `true`, то вместо введенных символов в текстовом поле будет отображаться «знак пароля», принятый в системе (например, точка или «звездочка»).

3.6.7 Событие `TextChanged`

Из всех событий элемента `TextBox` следует отметить событие `TextChanged`, которое «срабатывает» при изменении текста в элементе. Пусть в форме размещены текстовое поле, метка и требуется сделать так, чтобы при изменении текста в текстовом поле также менялся текст на метке:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        textBox1.TextChanged += text-
Box1_TextChanged;
    }
    private void textBox1_TextChanged(object sender, EventArgs e) {
        label1.Text = textBox1.Text;
    }
}
```

3.7 Элемент `MaskedTextBox`

Элемент класса `MaskedTextBox` представляет обычное текстовое поле, которое может контролировать ввод пользователя и проверять его «автоматически» на наличие ошибок. Чтобы контролировать вводимые в поле символы, надо задать маску. Для задания маски можно применять следующие символы:

0: Позволяет вводить только цифры;
 9: Позволяет вводить цифры и пробелы;
 #: Позволяет вводить цифры, пробелы и знаки '+' и '-';
 L: Позволяет вводить только буквенные символы;
 ?: Позволяет вводить дополнительные необязательные буквенные символы;
 A: Позволяет вводить буквенные и цифровые символы;
 .: Задает позицию разделителя целой и дробной части;
 ,: Используется для разделения разрядов в целой части числа;
 :: Используется в временных промежутках - разделяет часы, минуты и секунды;
 /: Используется для разделения полей дат;
 \$: Используется в качестве символа валюты.

Чтобы задать маску, необходимо установить свойство `Mask` элемента в окне свойств (`Properties`), при этом, в результате откроется окно для задания одного из стандартных шаблонов маски. Элемент класса `MaskedTextBox` представляет ряд свойств, которые можно использовать для управления вводом:

- `BeepOnError` при установке значения `true` подает звуковой сигнал при введении некорректного символа.
- `PromptChar` указывает на символ, который отображается в поле на месте ввода символов. По умолчанию стоит знак подчеркивания.
- `HidePromptOnLeave` при установке в `true` требует, чтобы при потере текстовым полем фокуса скрывались символы, указанные в `PromptChar`
- `AsciiOnly` при установке в значение `true`, позволяет вводить только ASCII-символы, то есть символы из диапазона A-Z и a-z.

Более подробно можно ознакомиться здесь <https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.forms.maskedtextbox?view=netframework-4.7.2>

3.8 Выражения и операции в C#

Любое *выражение* в языке C# состоит из *операндов* и *операций*. В Приложении Б показаны сгруппированные операции по приоритету операций.

Для контроля значений, получаемых при работе с числовыми выражениями, в C# предусмотрено использование контролируемого и неконтролируемого контекстов. *Контролируемый контекст* объявляется в форме **checked** операторный-блок, либо как операция **checked(выражение)**¹³. *Неконтролируемый контекст* объявляется в форме **unchecked** операторный-блок, либо как операция **unchecked(выражение)**¹⁴.

Арифметические операции +, -, *, /, % определены для всех числовых типов, причем для коротких целых типов компилятор выполняет неявное преобразование типов. Арифметические операции для типов с плавающей запятой не генерируют исключительных ситуаций при переполнении, потере точности или делении на ноль, но в результате таких операций получаются особые значения, определённые в виде констант **double.NaN**, **double.NegativeInfinity**, **double.PositiveInfinity** (т.е. «не число», «минус бесконечность», «плюс бесконечность»).

¹³ Если при вычислении в контролируемом контексте получается значение, выходящее за пределы целевого типа, то генерируется либо ошибка компиляции (для константных выражений), либо обрабатываемое исключение (для выражений с переменными)

¹⁴ При использовании неконтролируемого контекста выход за пределы целевого типа ведёт к автоматическому «урезанию» результата либо путём отбрасывания бит (целые типы), либо путём округления (вещественные типы). Неконтролируемый контекст применяется в вычислениях по умолчанию.

Методы пользовательских типов состоят из операторов, которые выполняются последовательно. В С# определен *операторный блок* – последовательность операторов, заключённая в фигурные скобки.

❑ Операторы объявления

К операторам объявления относятся *операторы объявления переменных* и *операторы объявления констант*. Для объявления локальных переменных метода применяется оператор следующего формата:

тип имя-переменной [= начальное-значение];

Здесь *тип* – тип переменной, *имя-переменной* – допустимый идентификатор, необязательное *начальное-значение* – литерал или выражение, соответствующее типу переменной.

Локальные переменные методов не могут использоваться в вычислениях, не будучи инициализированы.

Примеры объявления переменных:

```
int a; // простейший вариант объявления
int a = 10; // объявление с инициализацией
int a, b, c; // объявление однотипных переменных
int a = 10, b = 15; // инициализация нескольких переменных
```

Локальная переменная может быть объявлена без указания типа, с использованием ключевого слова **var**. В этом случае компилятор выводит тип переменной из обязательного выражения инициализации.

```
var x = 5;
var y = "Work";
var z = new Student();
```

Оператор объявления константы имеет следующий синтаксис:
const *тип-константы* *имя-константы* = *выражение*;

Допустимый *тип-константы* – это числовой тип, тип **bool**, тип **string**, перечисление или произвольный ссылочный тип. *Выражение*, которое присваивается константе, должно быть полностью вычислимо на момент компиляции. Обычно в качестве выражения используется литерал соответствующего типа. Для

ссылочных типов (за исключением `string`) единственно допустимым выражением является `null`. В одном операторе можно определить несколько однотипных констант:

```
const double Pi = 3.1415926, E = 2.718;
const string Name = "Work";
const object n = null;
```

Область доступа к переменной или константе ограничена операторным блоком, содержащим объявление: `{ int i = 10; }`

Если операторные блоки вложены друг в друга, то внутренний блок не может содержать объявлений переменных, идентификаторы которых совпадают с переменными внешнего блока.

□ *Операторы выражений*

Операторы выражений – это выражения, одновременно являющиеся допустимыми операторами:

- операция присваивания (включая инкремент и декремент);
- операция вызова метода или делегата;
- операция создания объекта.

Примеры:

```
x = 2 + 2; // присваивание
x++; // инкремент
Console.Write(x); // вызов метода
new Student(); // создание объекта
```

3.9 Ввод/вывод данных в программу

Для ввода данных чаще всего используют элемент управления `TextBox`, через обращение к его свойству `Text`. Свойство `Text` хранит в себе строку введенных символов. При вводе числовых данных необходимо перевести строку в целое или вещественное число. Для этого у типов существуют методы `Parse` для преобразования строк в числа:

```
private void button1_Click(object sender, EventArgs e) {
    string s = textBox1.Text;
    int a = int.Parse(s);
    int b = a * a;
}
```

Для преобразований предусмотрен специальный класс методов – класс `Convert` в пространстве имен `System`, который содержит 15 статических методов вида `ToType` (`ToBoolean`, ... `ToInt64`), где `Type` может принимать значения от `Boolean` до `UInt64`. Методы преобразования строк в числа (типа `double.Parse()` или `Convert.ToFloat()`) учитывают региональные настройки `Windows`¹⁵. Перед выводом числовые данные следует преобразовать назад в строку. Для этого у каждой переменной существует метод `ToString()`, который возвращает в результате строку с символьным представлением значения. Вывод данных можно осуществлять в элементы `TextBox` или `Label`, используя свойство `Text`. Например:

```
private void button1_Click(object sender, EventArgs e) {
    string s = textBox1.Text;
    int a = int.Parse(s);
    int b = a * a;
    label1.Text = b.ToString();
}
```

2.10 Стандартные математические функции

При вычислении математических операции используются методы класса `Math`. Этот класс содержит два статических поля, задающих константы `E` и `PI`, а также 23 статических метода:

- тригонометрические функции - `Sin`, `Cos`, `Tan`;
- обратные тригонометрические функции - `ASin`, `ACos`, `ATan`, `ATan2` (`sinx`, `cosx`);
- гиперболические функции – `Tanh`, `Sinh`, `Cosh`;
- экспоненту и логарифмические функции – `Exp`, `Log`, `Log10`;
- модуль, корень, знак – `Abs`, `Sqrt`, `Sign`;
- функции округления – `Ceiling`, `Floor`, `Round`;

¹⁵ Поэтому в полях `TextBox` в формах следует вводить дробные числа с запятой, а не с точкой. В противном случае преобразование не выполнится, а программа остановится с ошибкой.

- минимум, максимум, степень, остаток – Min, Max, Pow, IEEERemainder

В тригонометрических функциях все аргументы задаются в радианах.

Пример вызова математических функций:

```
private double m_longBase;
private double m_shortBase;
private double m_leftLeg;
private double m_rightLeg;
public MathTrapezoidSample(double longbase, double
shortbase, double leftLeg, double rightLeg){
    m_longBase = Math.Abs(longbase);
    m_shortBase = Math.Abs(shortbase);
    m_leftLeg = Math.Abs(leftLeg);
    m_rightLeg = Math.Abs(rightLeg);
}
private double GetRightSmallBase(){
    return (Math.Pow(m_rightLeg,2.0) -
Math.Pow(m_leftLeg,2.0) + Math.Pow(m_longBase,2.0) +
Math.Pow(m_shortBase,2.0) - 2* m_shortBase *
m_longBase) / (2*(m_longBase - m_shortBase));
}
public double GetHeight(){
    double x = GetRightSmallBase();
    return Math.Sqrt(Math.Pow(m_rightLeg,2.0) -
Math.Pow(x,2.0));
}
public double GetSquare(){
    return GetHeight() * m_longBase / 2.0;
}
public double GetLeftBaseRadianAngle(){
    double sinX = GetHeight()/m_leftLeg;
    return Math.Round(Math.Asin(sinX),2);
}
public double GetRightBaseRadianAngle() {
    double x = GetRightSmallBase();
    double cosX = (Math.Pow(m_rightLeg,2.0) +
Math.Pow(x,2.0) -
Math.Pow(GetHeight(),2.0))/(2*x*m_rightLeg);
```

```

return Math.Round(Math.Acos(cosX), 2);
}

```

Индивидуальные задания

По указанию преподавателя выберите индивидуальное задание. Уточните условие задания, количество, наименование, типы исходных данных. В соответствии с этим спроектируйте визуальную форму - установите необходимое количество TextBox, тексты заголовков на форме, размеры шрифтов, а также типы переменных и функции преобразования при вводе и выводе результатов. Создайте программу вычисления указанной величины. Для проверки правильности программы используйте пакет MathCad.

$$1. \quad t = \frac{2 \cos(x - \pi/6)}{0,5 + \sin^2 y} \left(1 + \frac{z^2}{3 - z^2/5} \right)$$

$$2. \quad u = \frac{\sqrt[3]{8 + |x - y|^2 + 1}}{x^2 + y^2 + 2} - e^{|x-y|} (tg^2 z + 1)^x$$

$$3. \quad v = \frac{1 + \sin^2(x + y)}{\left| x - \frac{2y}{1 + x^2 y^2} \right|} x^{|y|} + \cos^2[\arctg(1/z)]$$

$$4. \quad w = |\cos x - \cos y|^{(1+2\sin^2 y)} \left(1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4} \right)$$

$$5. \quad \alpha = \ln(y^{-\sqrt{|x|}})(x - y/2) + \sin^2 \arctg(z).$$

$$6. \quad \beta = \sqrt{10(\sqrt[3]{x} + x^{y+2})} \cdot (\arcsin^2 z - |x - y|)$$

$$7. \quad \gamma = 5 \arctg(x) - \frac{1}{4} \arccos(x) \frac{x + 3|x - y| + x^2}{|x - y|z + x^2}.$$

$$8. \quad \varphi = \frac{e^{|x-y|} |x-y|^{x+y}}{\arctg x + \arctg z} + \sqrt[3]{x^6 + \ln^2 y}.$$

$$9. \quad \psi = |x^{y/x} - \sqrt[3]{y/x}| + (y-x) \frac{\cos y - z/(y-x)}{1 + (y-x)^2}$$

$$10. \quad a = 2^{-x} \sqrt{x + \sqrt[4]{|y|}} \sqrt[3]{e^{x-1/\sin z}}.$$

$$11. \quad b = y^{\sqrt[3]{|x|}} + \cos^3 y \frac{|x-y| \cdot \left(1 + \frac{\sin^2 z}{\sqrt{x+y}}\right)}{e^{|x-y|} + x/2}.$$

$$12. \quad c = 2^{y^x} + (3^x)^y - \frac{y \cdot (\arctg z - \pi/6)}{|x| + \frac{1}{y^2 + 1}}.$$

$$13. \quad f = \frac{\sqrt[4]{y + \sqrt[3]{x-1}}}{|x-y| (\sin^2 z + \tg z)}$$

$$14. \quad g = \frac{y^{x+1}}{\sqrt[3]{|y-2|} + 3} + \frac{x+y/2}{2|x+y|} (x+1)^{-1/\sin z}$$

$$15. \quad h = \frac{x^{y+1} + e^{y-1}}{1 + x|y - \tg z|} (1 + |y-x|) + \frac{|y-x|^2}{2} - \frac{|y-x|^3}{3}$$

$$16. \quad y = 10^4 \frac{ax}{b^2} - \left| \frac{a-b}{kx} \right| + \frac{\ln 3}{\sqrt[3]{ax^2 + b^2}} - e^{-kx}$$

$$17. y = \cos k(x - a) + 10^{-4} \frac{(x + a)^3 + x^4 d}{k(x - a)^3} + \frac{\sqrt[5]{|x + a|}}{2.4b}.$$

$$18. y = 10^4 \sin^2 i - \frac{0.32x^3 + 4x + b}{\cos ia} \sqrt[6]{0.32x^3 - b} + |b|$$

Порядок выполнения практической работы

- 1) Запустите среду разработки Visual Studio.
- 2) Визуально спроектируйте форму основной программы в соответствии с индивидуальным заданием.
- 3) Выполните программирование события(ий) в программе.
- 4) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.
- 5) Выполните отчет по практической работе.

Контрольные вопросы

- 1) Как получить доступ к параметрам TextBox?
- 2) Как добавить текст?
- 3) Какие есть свойства у элемента TextBox?
- 4) Как сделать текстовую строку в виде пароля?
- 5) Как распределить текст на несколько строк?
- 6) Какие параметры имеет элемент Label?
- 7) Для чего нужны LinkLabel?
- 8) Каким образом обработать событие изменения текста в поле TextBox?

ПРАКТИЧЕСКАЯ РАБОТА № 4

Тема: «Работа с элементами ввода/вывода информации в Windows приложения»

Цель работы – Изучить систему ввода-вывода, получить навыки работы с компонентами выбора и списками.

Теоретические сведения

4.1 Операторы C#

□ Операторы перехода

К операторам перехода относятся `break`, `continue`, `goto`, `return`, `throw`. Оператор `break` используется для выхода из операторного блока циклов и оператора `switch`. Оператор `break` выполняет переход к оператору, расположенному за блоком. Оператор `continue` располагается в теле цикла и применяется для запуска новой итерации цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается `continue`.

Оператор `goto` передаёт управление на помеченный оператор. Обычно данный оператор употребляется в форме `goto метка`, где *метка* – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
```

```
...
```

```
label:
```

```
A = 100;
```

Возможно использование оператора `goto` в одной из следующих форм при применении оператора `switch`:

```
goto case константа;
```

```
goto default;
```

Оператор **return** служит для завершения методов. Оператор **throw** генерирует исключительную ситуацию.

❑ Операторы выбора

Операторы выбора – это операторы **if** и **switch**. Оператор **if** в языке C# имеет следующий синтаксис:

```
if (условие)
    вложенный-оператор-1
[else
    вложенный-оператор-2]
```

Здесь *условие* – это некоторое булево выражение, *вложенный-оператор* – оператор (за исключением оператора объявления) или операторный блок. Ветвь **else** является необязательной.

Оператор **switch** выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора **switch**:

```
switch (выражение)
{
    case константное-выражение-1:
        операторы
        оператор-перехода
    case константное-выражение-2:
        операторы
        оператор-перехода
    ...
    [default:
        операторы
        оператор-перехода]
```

Выражение должно возвращать значение целочисленного типа (включая **char**), булево значение, строку или элемент перечисления. При совпадении тестируемого и константного выражений выполняется соответствующая ветвь **case**. Если совпаде-

ния не обнаружено, то выполняется ветвь **default** (если она есть). *оператор-перехода* – это один из следующих операторов: **break**, **goto**, **return**, **throw**. Оператор **goto** используется с указанием либо ветви **default** (**goto default**), либо определённой ветви **case** (**goto case константное-выражение**).

Хотя после **case** может быть указано только одно константное выражение, при необходимости несколько ветвей **case** можно сгруппировать следующим образом:

```
switch (n)
{
    case 0:
    case 1:
    case 2:
        ...
}
```

❑ Операторы циклов

К *операторам циклов* относятся операторы **for**, **while**, **do-while**, **foreach**. Для циклов с известным числом итераций используется оператор **for**:

for (*[инициализатор]*; *[условие]*; *[итератор]*) *вложенный-оператор*

Здесь *инициализатор* задаёт начальное значение счётчика (или счётчиков) цикла. Для счётчика может использоваться существующая переменная или объявляться новая переменная, время жизни которой будет ограничено циклом (при этом вместо типа переменной допустимо указать **var**). Цикл выполняется, пока булево *условие* истинно, а *итератор* определяет изменение счётчика цикла на каждой итерации.

Простейший пример использования цикла **for**:

```
for (int i = 0; i < 10; i++)//i доступна только в цикле for
    Console.WriteLine(i);// вывод чисел от 0 до 9
```

В инициализаторе можно объявить и задать начальные значения для нескольких счётчиков одного типа. В этом случае итератор может представлять собой последовательность из нескольких операторов, разделённых запятой:

```
// цикл выполнится 5 раз, на последней итерации i = 4, j = 6
for (int i = 0, j = 10; i < j; i++, j--)
    Console.WriteLine("i = {0}, j = {1}", i, j);
```

Если число итераций цикла заранее неизвестно, можно использовать цикл **while** или цикл **do-while**. Данные циклы имеют схожий синтаксис:

while (*условие*) *вложенный-оператор*

do
вложенный-оператор
while (*условие*);

В обоих оператора цикла тело цикла выполняется, пока *булево условие* истинно.

Для перебора элементов объектов перечисляемых типов, таких как массивы в C# применяется цикл **foreach**:

foreach (*тип идентификатор in коллекция*) *вложенный-оператор*

В заголовке цикла объявляется переменная, которая будет последовательно принимать значения элементов коллекции. Вместо указания типа этой переменной можно использовать ключевое слово **var**.

❑ Прочие операторы

К группе прочих операторов относятся операторы **lock** и **using**. Первый связан с синхронизацией потоков выполнения, второй – с процессом освобождения локальной переменной.

4.2 Сведения о массивах в C#

Массивы – это ссылочные пользовательские типы. Объявление массива в C# схоже с объявлением переменной, но после

указания типа размещается *спецификатор размерности* – пара квадратных скобок:

```
int[] data;
```

Массив является ссылочным типом, поэтому перед началом работы любой массив должен быть создан в памяти. Для этого используется конструктор в форме `new тип[количество-элементов]`.

```
int[] data;
```

```
data = new int[10];
```

Создание массива можно совместить с его объявлением:

```
int[] data = new int[10];
```

Созданный массив автоматически заполняется значениями по умолчанию для своего базового типа (ссылочные типы – `null`, числа – 0, тип `bool` – `false`).

Для доступа к элементу массива указывается имя массива и индекс в квадратных скобках: `data[0] = 10`. Индекс массива должен неявно приводиться к типам `int`, `uint`, `long` или `ulong`. Элементы массива нумеруются с нуля, в С# не предусмотрено синтаксических конструкций для указания особого значения нижней границы массива. При выходе индекса массива за допустимый диапазон генерируется исключительная ситуация.

В С# существует способ задания всех элементов массива при создании. Для этого используется список значений в фигурных скобках. При этом можно не указывать количество элементов, а также полностью опустить указание на тип и ключевое слово `new`:

```
int[] data_1 = new int[4] { 1, 2, 3, 5};
```

```
int[] data_2 = new int[] { 1, 2, 3, 5};
```

```
int[] data_3 = new[] { 1, 2, 3, 5};
```

```
int[] data_4 = { 1, 2, 3, 5};
```

Первые три примера инициализации допускают указание вместо типа переменной ключевого слова `var` (например, `var`

`data_3 = new[] { 1, 2, 3, 5 };`). Компилятор вычислит тип массива автоматически.

При необходимости можно объявить массивы, имеющие несколько размерностей. Для этого в спецификаторе размерности помещают запятые, «разделяющие» размерности:

```
// двумерный массив d
int[,] d;
d = new int[10, 2];
// трёхмерный массив Cube
int[, ,] Cube = new int[3, 2, 5];
// объявим двумерный массив и инициализируем его
int[,] c = new int[2, 4] {
    { 1, 2, 3, 4 },
    { 10, 20, 30, 40 }
};
// то же самое, но короче
int[,] c = { { 1, 2, 3, 4 }, { 10, 20, 30, 40 } };
```

В приведённых примерах объявлялись массивы из нескольких размерностей. Такие массивы всегда являются прямоугольными. Можно объявить *массив массивов*, используя следующий синтаксис:

```
int[][] table; // table – массив одномерных массивов
table = new int[2][]; // в table будет 2 одномерных массива
table[0] = new int[2]; // в первом массиве будет 2 элемента
table[1] = new int[20]; // во втором – 20 элементов
table[1][3] = 1000; // работаем с элементами table
// совместим объявление и инициализацию массива массивов
int[][] t = { new[] { 10, 20 }, new[] { 1, 2, 3 } };
```

Язык C# содержит специальные инструкции для работы с одномерными массивами, индексированными с нуля. Поэтому массив массивов обрабатывается немного быстрее, чем двумерный массив.

При работе с массивом можно использовать цикл `foreach`, перебирающий все элементы. В цикле `foreach` возможно перемещение по массиву в одном направлении — от начала к концу, при этом попытки присвоить значение элементу массива игнорируются. В следующем фрагменте кода производится суммирование элементов массива:

```
int[] data = { 1, 3, 5, 7, 9 };
var sum = 0;
foreach (var element in data) {
    sum += element;
}
```

Массивы *ковариантны* для ссылочных типов. Это означает следующее: если ссылочный тип А неявно приводим к ссылочному типу В, массив с элементами типа А может быть присвоен массиву с элементами типа В. При этом количество элементов в массиве роли не играет, но массивы должны иметь одинаковую размерность.

```
public class Student { ... } // объявление класса
Student[] students = new Student[10];
object[] array = students; // ковариантность массивов
```

Все массивы в платформе .NET могут рассматриваться как классы, являющиеся потомками класса `System.Array`. Возможности данного класса описаны в рамке рассмотрения коллекций.

4.3 Компонент класса `ListBox`.

Элемент `ListBox` (рис. 4.1) представляет собой простой список. Ключевым свойством этого элемента является свойство **Items**, которое как раз и хранит набор всех элементов списка. Элементы в список могут добавляться как во время разработки, так и программным способом. В **Visual Studio** в окне **Properties** (Свойства) для элемента `ListBox` мы можем найти свойство **Items**.

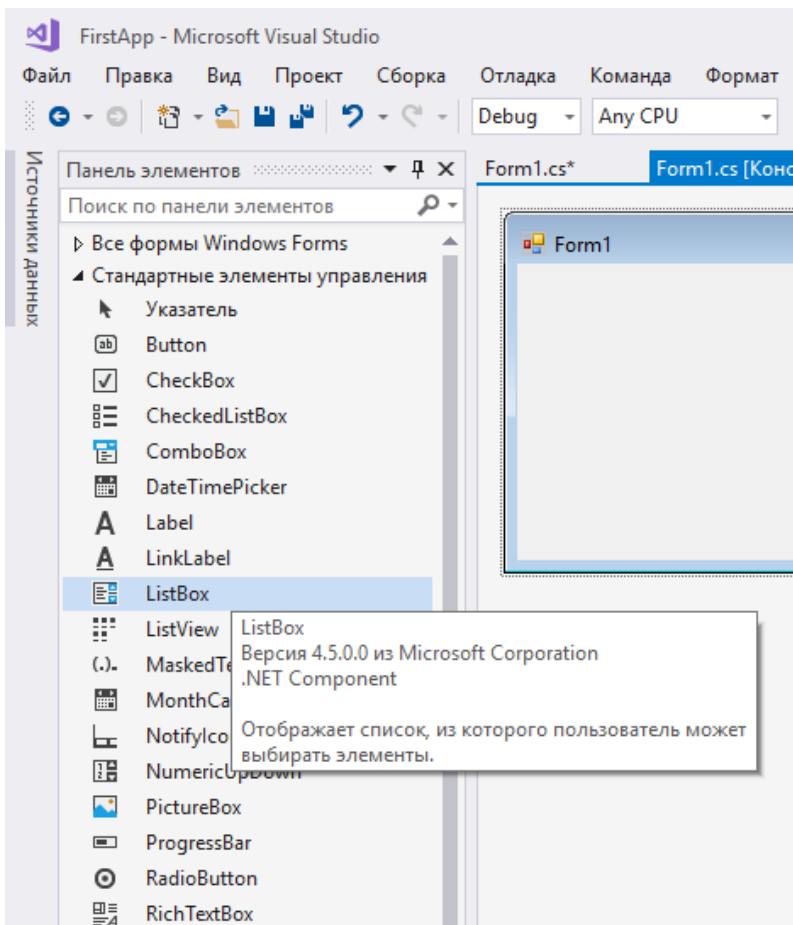


Рис. 4.1 — Компонент ListBox в панели элементов.

После двойного щелчка на свойство нам отобразится окно для добавления элементов в список (рис 4.2). В пустое поле мы вводим по одному элементу списка - по одному на каждой строке. После этого все добавленные нами элементы окажутся в списке, и мы сможем ими управлять (рис. 4.3).

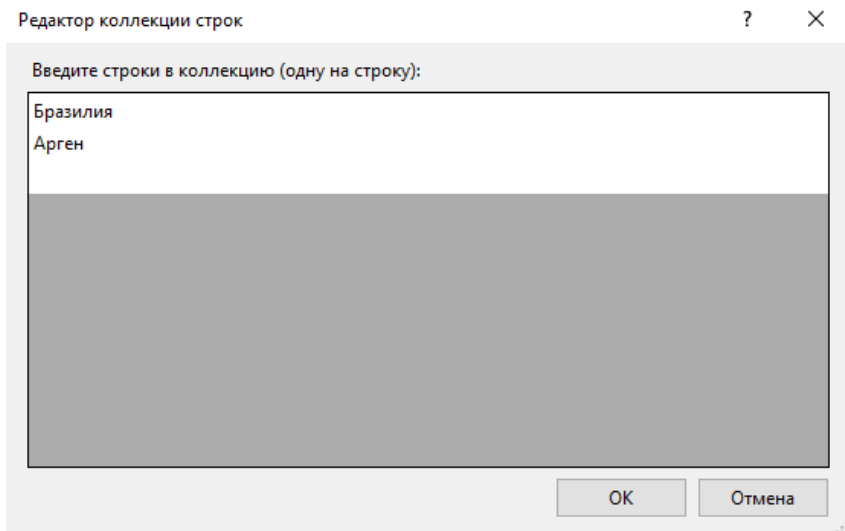


Рис. 4.2 — Изменение свойства для элемента ListBox

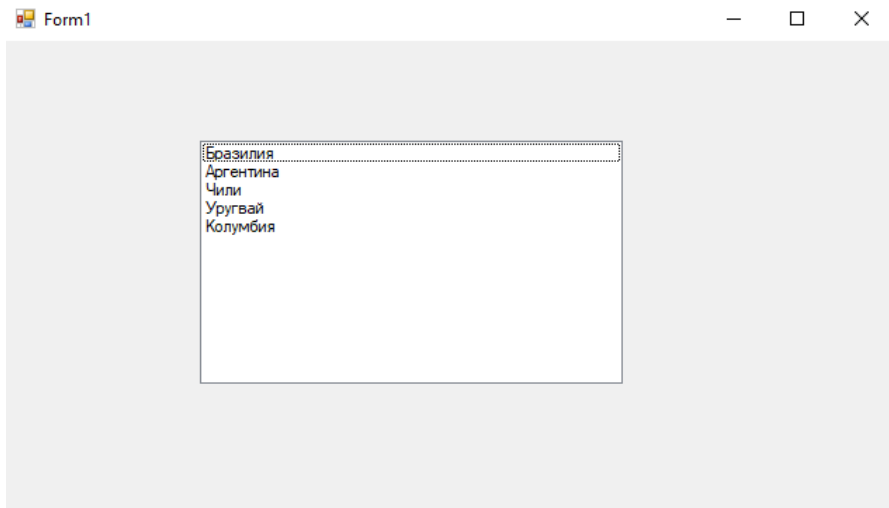


Рис. 4.3 — Элемент управления ListBox

4.4 Программное управление элементами в `ListBox`

4.4.1 Добавление элементов

Итак, все элементы списка входят в свойство `Items`, которое представляет собой коллекцию. Для добавления нового элемента в эту коллекцию, а значит и в список, надо использовать метод `Add`, например: `listBox1.Items.Add("Новый элемент");`

При использовании этого метода каждый добавляемый элемент добавляется в конец списка. Можно добавить сразу несколько элементов, например, массив. Для этого используется метод `AddRange`:

```
string[] countries = { "Бразилия", "Аргентина", "Чили",  
"Уругвай", "Колумбия" };  
listBox1.Items.AddRange(countries);
```

4.4.2 Вставка элементов

В отличие от простого добавления вставка производится по определенному индексу списка с помощью метода `Insert`:

```
listBox1.Items.Insert(1, "Парагвай");
```

В данном случае вставляем элемент на вторую позицию в списке, так как отсчет позиций начинается с нуля.

4.4.3 Удаление элементов

Для удаления элемента по его тексту используется метод `Remove`: `listBox1.Items.Remove("Чили");`

Чтобы удалить элемент по его индексу в списке, используется метод `RemoveAt`: `listBox1.Items.RemoveAt(1);`

Кроме того, можно очистить сразу весь список, применив метод `Clear`: `listBox1.Items.Clear();`

4.4.4 Доступ к элементам списка

Используя индекс элемента, можно сам элемент в списке. Например, получим первый элемент списка:

```
string firstElement = listBox1.Items[0];
```

Метод `Count` позволяет определить количество элементов в списке: `int number = listBox1.Items.Count();`

4.4.5 Выделение элементов списка

При выделении элементов списка возможно управление ими как через индекс, так и через сам выделенный элемент. Получить выделенные элементы можно с помощью следующих свойств элемента **ListBox**:

- **SelectedIndex**: возвращает или устанавливает номер выделенного элемента списка. Если выделенные элементы отсутствуют, тогда свойство имеет значение -1
- **SelectedIndices**: возвращает или устанавливает коллекцию выделенных элементов в виде набора их индексов
- **SelectedItem**: возвращает или устанавливает текст выделенного элемента
- **SelectedItems**: возвращает или устанавливает выделенные элементы в виде коллекции

По умолчанию список поддерживает выделение одного элемента. Чтобы добавить возможность выделения нескольких элементов, надо установить у его свойства `SelectionMode` значение `MultiSimple`.

Чтобы выделить элемент программно, надо применить метод `SetSelected(int index, bool value)`, где `index` - номер выделенного элемента. Если второй параметр - `value` имеет значение `true`, то элемент по указанному индексу выделяется, если `false`, то выделение наоборот скрывается:

```
listBox1.SetSelected(2, true); // будет выделен третий элемент
```

Чтобы снять выделение со всех выделенных элементов, используется метод `ClearSelected`.

Из всех событий элемента **ListBox** надо отметить в первую очередь событие `SelectedIndexChanged`, которое возникает при изменении выделенного элемента:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApp6
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            string[] countries = { "Бразилия", "Аргентина",
"Чили", "Уругвай", "Колумбия" };
            listBox1.Items.AddRange(countries);
            listBox1.SelectedIndexChanged +=
listBox1_SelectedIndexChanged;
        }
        void listBox1_SelectedIndexChanged(object sender,
EventArgs e)
        {
            string selectedCountry =
listBox1.SelectedItem.ToString();
            MessageBox.Show(selectedCountry);
        }
    }
}

```

В данном случае по выбору элемента списка будет отображаться сообщение с выделенным элементом.

4.5 Включение сетки

В режиме создания **Windows Forms** по умолчанию сетка для выравнивания элементов отключена. Чтобы ее включить, нужно зайти в настройки: **Tools -> Options... -> Windows Forms Designer -> General** (Средства -> Параметры -> Конструктор **Windows Forms -> Общие**) – рис. 4.4 и установить **Layout mode** (Режим макета) в значение **SnapToGrid**, **Show grid: true**, **Snap to Grid: true**. На рисунке 4.5 показан внешний вид сетки.

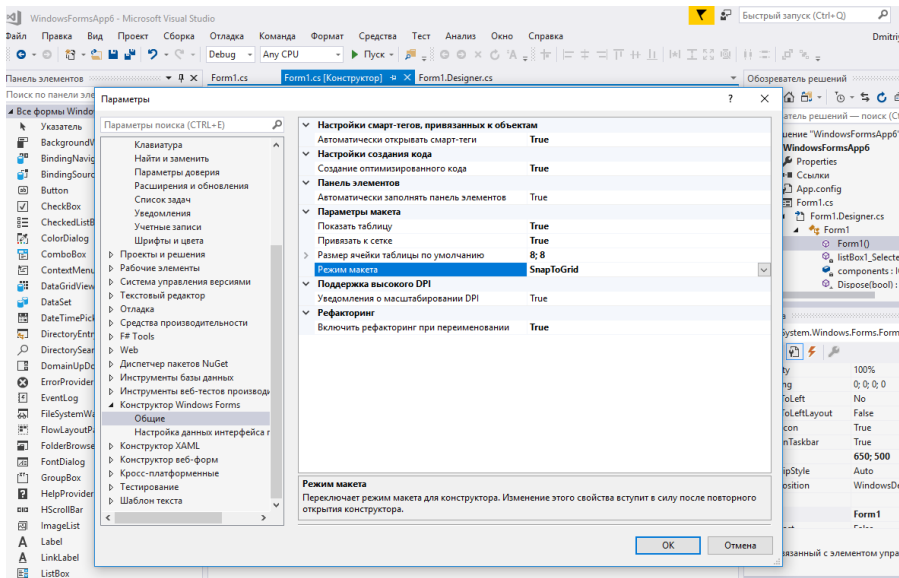


Рис. 4.4 — Установка сетки в настройках Visual Studio 2017

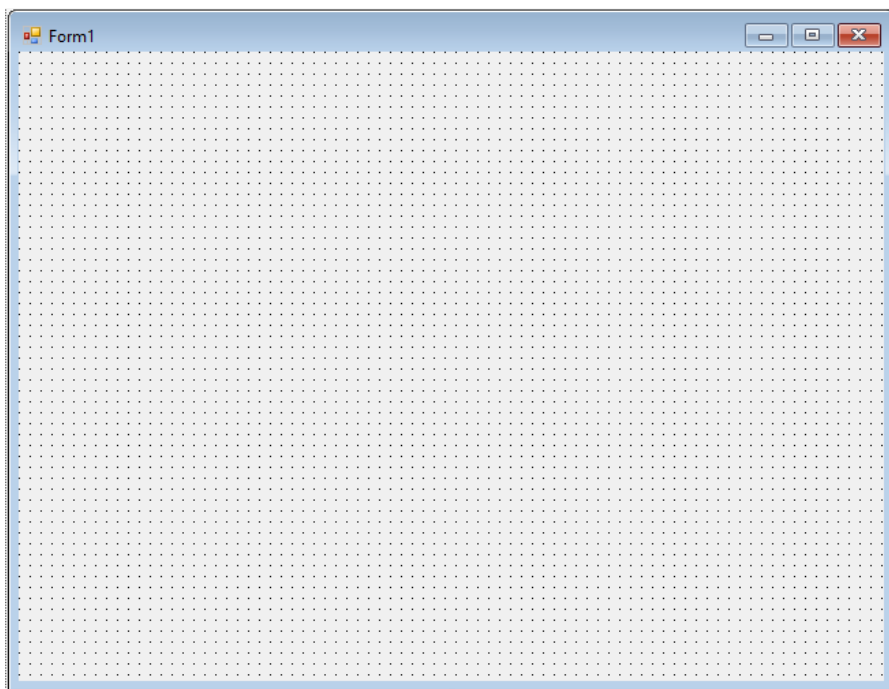


Рис. 4.5 — Внешний вид сетки

4.6 Элементы управления **CheckBox** и **RadioButton**

Элемент **CheckBox** или флажок предназначен для установки одного из двух значений: отмечен или не отмечен. Чтобы отметить флажок, надо установить у его свойства **Checked** значение `true`.

Кроме свойства `Checked` у элемента **CheckBox** имеется свойство `CheckState`, которое позволяет задать для флажка одно из трех состояний - `Checked` (отмечен), `Indeterminate` (флажок не определен - отмечен, но находится в неактивном состоянии) и `Unchecked` (не отмечен)

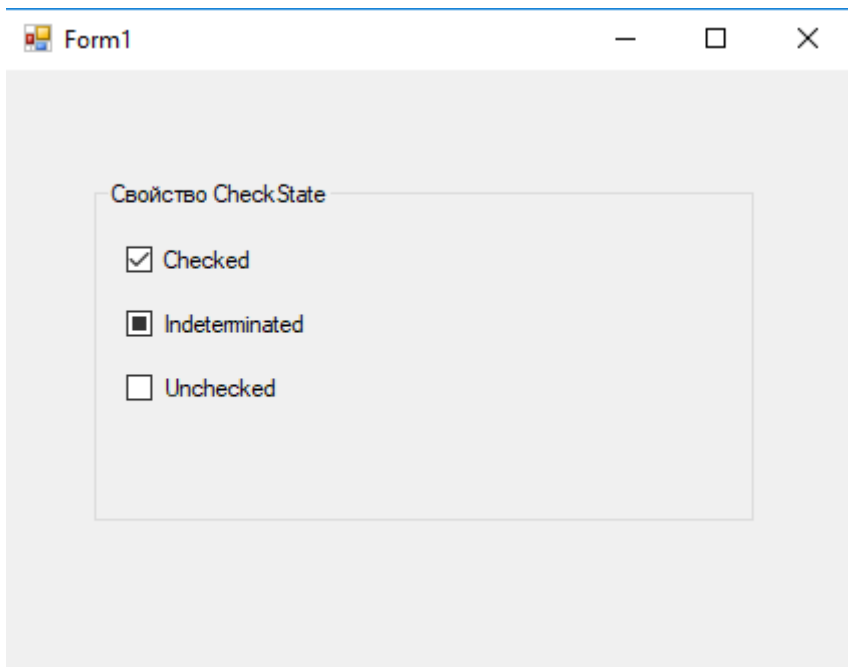


Рис. 4.6 — Внешний вид RadioButton

Также следует отметить свойство `AutoCheck` - если оно имеет значение `false`, то мы не можем изменять состояние флажка. По умолчанию оно имеет значение `true`.

При изменении состояния флажка он генерирует событие `CheckedChanged`. Обработывая это событие, мы можем получать измененный флажок и производить определенные действия:


```

private void checkBox1_CheckedChanged(object sender, EventArgs
e) {
    CheckBox checkBox = (CheckBox)sender; // приводим отправителя
    //к элементу типа CheckBox
    if (checkBox.Checked == true) {
        MessageBox.Show("Флажок " + checkBox.Text + " теперь
отмечен");
    } else {
        MessageBox.Show("Флажок " + checkBox.Text + " теперь не
отмечен");
    }
}
}

```

Элемент **RadioButton** или переключатель (рис. 4.7) похож на элемент **CheckBox**. Переключатели располагаются группами, и включение одного переключателя означает отключение всех остальных.

Чтобы установить у переключателя включенное состояние, надо присвоить его свойству `Checked` значение `true`.

Для создания группы переключателей, из которых можно бы было выбирать, надо поместить несколько переключателей в какой-нибудь контейнер, например, в элементы **GroupBox** или **Panel**. Переключатели, находящиеся в разных контейнерах, будут относиться к разным группам. Похожим образом можно перехватывать переключение переключателей в группе, обрабатывая событие `CheckedChanged`.

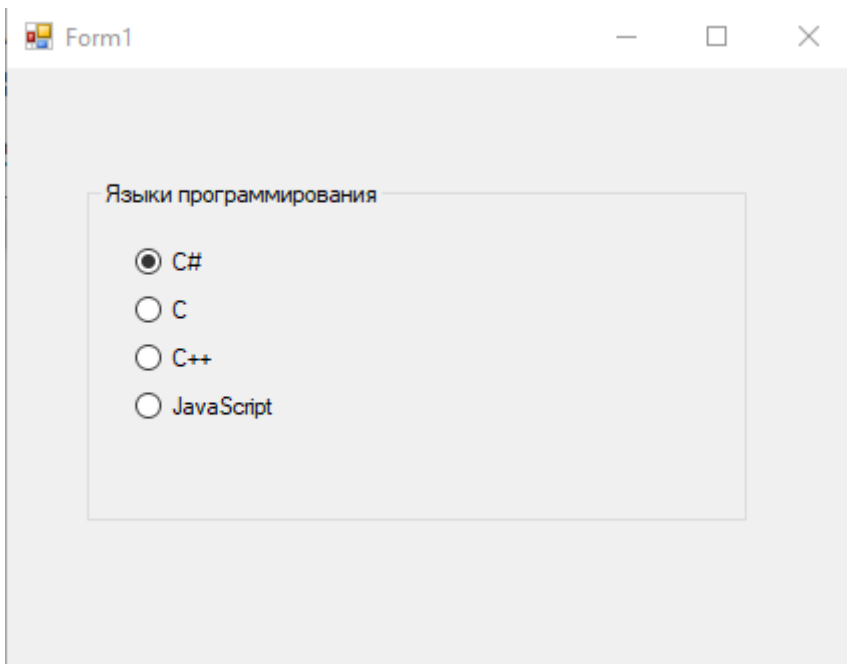


Рис. 4.7 — Внешний вид **RadioButton**

Связав каждый переключатель группы с одним обработчиком данного события, мы сможем получить тот переключатель, который в данный момент выбран:

```
private void radioButton_CheckedChanged(object sender, EventArgs e) {  
    // приводим отправителя к элементу типа RadioButton  
    RadioButton radioButton = (RadioButton)sender;  
    if (radioButton.Checked) {  
        MessageBox.Show("Вы выбрали " + radioButton.Text);  
    }  
}
```

Индивидуальные задания

Выберите индивидуальное задание из нижеприведенного списка согласно номеру в подгруппе. В качестве $f(x)$ использовать по выбору: $sh(x)$, x^2 , e^x . Результат очередного вычисления добавляйте в компонент `ListBox`, в интерфейсе предусмотрите кнопку очистки старых результатов.

$$1 \quad a = \begin{cases} (f(x) + y)^2 - \sqrt{f(x)y}, & xy > 0 \\ (f(x) + y)^2 + \sqrt{|f(x)y|}, & xy < 0 \\ (f(x) + y)^2 + 1, & xy = 0. \end{cases}$$

$$2 \quad b = \begin{cases} \ln(f(x)) + (f(x)^2 + y)^3, & x/y > 0 \\ \ln|f(x)/y| + (f(x) + y)^3, & x/y < 0 \\ (f(x)^2 + y)^3, & x = 0 \\ 0, & y = 0. \end{cases}$$

$$3 \quad c = \begin{cases} f(x)^2 + y^2 + \sin(y), & x - y = 0 \\ (f(x) - y)^2 + \cos(y), & x - y > 0 \\ (y - f(x))^2 + \operatorname{tg}(y), & x - y < 0. \end{cases}$$

$$4 \quad d = \begin{cases} (f(x) - y)^3 + \operatorname{arctg}(f(x)), & x > y \\ (y - f(x))^3 + \operatorname{arctg}(f(x)), & y > x \\ (y + f(x))^3 + 0.5, & y = x. \end{cases}$$

$$5 \quad e = \begin{cases} i\sqrt{f(x)}, & i - \text{нечетное, } x \rangle 0 \\ i / 2\sqrt{|f(x)|}, & i - \text{четное, } x \langle 0 \\ \sqrt{|if(x)|}, & \text{иначе.} \end{cases}$$

$$6 \quad g = \begin{cases} e^{f(x)-|b|}, & 0.5 \langle xb \langle 10 \\ \sqrt{|f(x) + b|}, & 0.1 \langle xb \langle 0.5 \\ 2f(x)^2, & \text{иначе.} \end{cases}$$

$$7 \quad s = \begin{cases} e^{f(x)}, & 1 \langle xb \langle 10 \\ \sqrt{|f(x) + 4 * b|}, & 12 \langle xb \langle 40 \\ bf(x)^2, & \text{иначе.} \end{cases}$$

$$8 \quad j = \begin{cases} \sin(5f(x) + 3m|f(x)|), & -1 \langle m \langle x \\ \cos(3f(x) + 5m|f(x)|), & x \rangle m \\ (f(x) + m)^2, & x = m. \end{cases}$$

$$9 \quad l = \begin{cases} 2f(x)^3 + 3p^2, & x \rangle |p| \\ |f(x) - p|, & 3 \langle x \langle |p| \\ (f(x) - p)^2, & x = |p|. \end{cases}$$

$$10 \quad k = \begin{cases} \ln(|f(x)| + |q|), & |xq| \rangle 10 \\ e^{f(x)+q}, & |xq| \langle 10 \\ f(x) + q, & |xq| = 10 \end{cases}$$

$$\begin{aligned}
 11 \quad m &= \frac{\max(f(x), y, z)}{\min(f(x), y)} + 5. \\
 12 \quad n &= \frac{\min(f(x) + y, y - z)}{\max(f(x), y)}. \\
 13 \quad p &= \frac{|\min(f(x), y) - \max(y, z)|}{2}. \\
 14 \quad q &= \frac{\max(f(x) + y + z, xyz)}{\min(f(x) + y + z, xyz)}. \\
 15 \quad r &= \max(\min(f(x), y), z).
 \end{aligned}$$

Порядок выполнения лабораторной работы

- 1) Запустите среду разработки Visual Studio.
- 2) Визуально спроектируйте форму основной программы в соответствии с индивидуальным заданием.
- 3) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.
- 4) Выполните отчет по лабораторной работе.

Контрольные вопросы

- 1) Какие операции используются в C#?
- 2) Поясните особенность работы с операторов выбора.
- 3) Поясните особенность работы с операторов цикла.
- 4) Что такое массивы? Какие типы массивов?
- 5) Как можно инициализировать массив?
- 6) Какой используют цикл для работы с массивами?
- 7) Что такое компонент ListBox?
- 8) Поясните возможности применения ListBox?
- 9) Поясните назначение, сходство и отличие компонентов CheckBox и RadioButton?

ПРИЛОЖЕНИЕ А

Типы данных C#

| Логический тип | | | |
|---------------------------------------|-----------------------------|--|---------------------|
| Имя типа | Имя типа в CLR | Значения | Размер |
| <code>bool</code> | <code>System.Boolean</code> | true, false | 8 бит |
| Арифметические целочисленные типы | | | |
| Имя типа | Системный тип | Диапазон | Размер |
| <code>sbyte</code> | <code>System.SByte</code> | $-128 \dots -128$ | Знаковое, 8 бит |
| <code>byte</code> | <code>System.Byte</code> | $0 \dots 255$ | Беззнаковое, 8 бит |
| <code>short</code> | <code>System.Int16</code> | $-32768 \dots -32767$ | Знаковое, 16 бит |
| <code>ushort</code> | <code>System.UInt16</code> | $0 \dots 65535$ | Беззнаковое, 16 бит |
| <code>int</code> | <code>System.Int32</code> | $\approx (-2 \cdot 10^9 \dots -2 \cdot 10^9)$ | Знаковое, 32 бит |
| <code>uint</code> | <code>System.UInt32</code> | $\approx (0 \dots -4 \cdot 10^9)$ | Беззнаковое, 32 бит |
| <code>long</code> | <code>System.Int64</code> | $\approx (-9 \cdot 10^{18} \dots 9 \cdot 10^{18})$ | Знаковое, 64 бит |
| <code>ulong</code> | <code>System.UInt64</code> | $\approx (0 \dots 18 \cdot 10^{18})$ | Беззнаковое, 64 бит |
| Арифметический тип с плавающей точкой | | | |
| Имя типа | Имя типа в CLR | Диапазон | Точность |
| <code>float</code> | <code>System.Single</code> | $-1.5 \cdot 10^{-45} \dots +3.4 \cdot 10^{38}$ | 7 цифр |

| | | | |
|---|----------------|---|--------------------------|
| double | System.Double | $-5.0 \cdot 10^{-324} \dots$ $+1.7 \cdot 10^{308}$ | 15–16 цифр |
| Арифметический тип с фиксированной точкой | | | |
| Имя типа | Имя типа в CLR | Диапазон | Точность |
| decimal | System.Decimal | $-1.0 \cdot 10^{-28} \dots$ $+7.9 \cdot 10^{28}$ | 28–29 зна- чащих цифр |
| Символьные типы | | | |
| Имя типа | Имя типа в CLR | Диапазон | Точность |
| char | System.Char | U+0000 – U+ffff | Unicode символ |
| string | System.String | Строка из символов Unicode | |
| Объектный тип | | | |
| Имя типа | Системный тип | Примечание | |
| Object | System.Object | Прародитель всех типов | |

ПРИЛОЖЕНИЕ Б

Операции C# по приоритету

Таблица Б

| Категория | Expression | Описание |
|-----------|------------------------------|---|
| Первичный | <code>x.m</code> | Метод доступа к члену |
| | <code>x(...)</code> | Вызов методов и делегатов |
| | <code>x[...]</code> | Доступ к массиву и индексатору |
| | <code>x++</code> | Постфиксный инкремент |
| | <code>x--</code> | Постфиксный декремент |
| | <code>new T(...)</code> | Создание объекта и делегата |
| | <code>new T(...){...}</code> | Создание объекта с использованием инициализатора |
| | <code>new {...}</code> | Инициализатор анонимного объекта |
| | <code>new T[...]</code> | Создание массива |
| | <code>typeof(T)</code> | Получение объекта <code>System.Type</code> для <code>T</code> |

Продолжение таблицы Б

| Категория | Expression | Описание |
|-------------------|-----------------------------|---|
| Первичный | <code>checked(x)</code> | Вычисление выражения в контексте <code>checked</code> |
| | <code>unchecked(x)</code> | Вычисление выражения в контексте <code>unchecked</code> |
| | <code>default(T)</code> | Получение значения по умолчанию типа <code>T</code> |
| | <code>delegate {...}</code> | Анонимная функция (анонимный метод) |
| Унарный | <code>+x</code> | Идентификация |
| | <code>-x</code> | Отрицание |
| | <code>!x</code> | Логическое отрицание |
| | <code>~x</code> | Побитовое отрицание |
| | <code>++x</code> | Префиксный инкремент |
| | <code>--x</code> | Префиксный декремент |
| | <code>(T)x</code> | Явное преобразование <code>x</code> к типу <code>T</code> |
| Мультипликативный | <code>x * y</code> | Умножение |
| | <code>x / y</code> | Деление |
| | <code>x % y</code> | Остаток |
| Аддитивный | <code>x + y</code> | Сложение, объединение строк, объединение делегатов |
| | <code>x - y</code> | Вычитание, удаление делегата |
| Сдвиг | <code>x << y</code> | Поразрядный сдвиг влево |
| | <code>x >> y</code> | Поразрядный сдвиг вправо |

Продолжение таблицы Б

| Категория | Expression | Описание |
|---------------------------|----------------------|--|
| Отношение и проверка типа | $x < y$ | Меньше |
| | $x > y$ | Больше |
| | $x \leq y$ | Меньше или равно |
| | $x \geq y$ | Больше или равно |
| | $x \text{ is } T$ | Возвращает true , если x имеет тип T ; в противном случае возвращает false |
| | $x \text{ as } T$ | Возвращает значение x , типизированное как T , или null , если x имеет тип, отличный от T |
| Равенство | $x == y$ | Равно |
| | $x != y$ | Не равно |
| Логическое И | $x \& y$ | Целое побитовое И, логическое И |
| Исключающее ИЛИ | $x \wedge y$ | Целое побитовое исключающее ИЛИ, логическое исключающее ИЛИ |
| Логическое ИЛИ | $x \mid y$ | Целое побитовое ИЛИ, логическое ИЛИ |
| Условное И | $x \&\& y$ | y вычисляется только в том случае, если x имеет значение true |
| Условное ИЛИ | $x \mid\mid y$ | y вычисляется только в том случае, если x имеет значение false |
| Объединение с null | $x \text{ ?? } y$ | Если x имеет значение null , вычисляется y ; в противном случае вычисляется x |
| Условный | $x \text{ ? } y : z$ | Если x имеет значение true , вычисляется y ; если x имеет значение false , вычисляется z |

Продолжение таблицы Б

| Категория | Expression | Описание |
|------------------------------------|-----------------------------|--|
| Присваивание или анонимная функция | $x = y$ | Присваивание |
| | $x \text{ op} = y$ | Составное присваивание; поддерживаются следующие операторы: $* = / = \% = + = - = < < = > > = \& = \wedge = =$ |
| | $(\lambda x) \Rightarrow y$ | Анонимная функция (лямбда-выражение) |