

Московский государственный технический университет
имени Н. Э. Баумана

С. Н. Банников

РАЗРАБОТКА ПРОСТОГО КАЛЬКУЛЯТОРА НА ЯЗЫКЕ C#

*Методические указания к лабораторной работе по курсу
«Разработка приложений на языке C#»*

Москва

Издательство МГТУ им. Н. Э. Баумана

2016

УДК xxx.xx.x
ББК xx.xx
XNN

Рецензент

Банников С.Н.

Разработка простого калькулятора на языке C# : метод. указания к лабораторным работам по курсу «Разработка приложений на языке C#» / С. Н. Банников – М.: Изд-во МГТУ им. Н. Э. Баумана, 2016. – 134 с.: ил.

Изложены основы программирования на языке C# на примере создания простого калькулятора на языке C# в виде графического приложения на платформе Microsoft.NET Framework с использованием библиотеки WinForms. Дано описание синтаксиса языка в объеме, необходимом для реализации этого приложения. Приведена инструкция по работе с системой контроля версий исходных текстов TFS. Третий и четвертый разделы книги содержат справочные материалы по языку C#.

Для студентов, обучающихся по специальности «Компьютерные системы, комплексы и сети».

УДК xxx.xx.x
ББК xx.xx

Учебное издание

Банников Сергей Николаевич

Разработка простого калькулятора на языке C#

Редактор, Корректор

Компьютерная верстка

Подписано в печать xx.xx.xxxx. Формат

Усл. печ. л. Тираж 0 экз. Изд №. Заказ.

Издательство МГТУ им. Н. Э. Баумана

Типография МГТУ им. Н. Э. Баумана

105005, Москва, 2-я Бауманская ул., 5

© МГТУ им. Н. Э. Баумана, 2016

ВВЕДЕНИЕ

Язык программирования C# является достаточно востребованным среди сообщества разработчиков программного обеспечения во всем мире. По различным оценкам (RedMonk, IEEE Spectrum, TIOBE Software), язык C# занимает от 4-го до 6-го места, в процентном отношении это составляет от 5% до 30%. Разработчики для платформы Microsoft.NET Framework пользуются спросом на рынке труда, а язык C# является основным языком программирования для данной платформы.

Целями данной лабораторной работы являются:

1. Знакомство со средой разработки Microsoft Visual Studio
2. Знакомство с системой контроля версий Team Foundation Service
3. Знакомство с базовыми синтаксическими элементами языка C#
4. Знакомство с базовыми возможностями библиотеки Windows Forms
5. Создание простого приложения – калькулятора.

1. Система контроля версий TFS

1.1. Общие сведения

При выполнении лабораторных работ по курсу «Разработка приложений на языке C#» используется система контроля версий исходных текстов программ Microsoft Team Foundation Server (TFS). Система контроля версий позволяет выполнять сравнение различных вариантов (версий) исходных текстов, ведет историю изменения всех файлов и поддерживает совместную работу над текстами программ. Настройка подключения к TFS осуществляется в два этапа – подключение к portalу TFS и настройка среды разработки Visual Studio.

1.2. Подключение к portalу

Внешнее подключение к TFS осуществляется по протоколу HTTPS. Для настройки соединения следует установить сертификат сервера (файл **lab.croc.ru.cer**). Его следует загрузить из рабочей группы Jive и сохранить локально на компьютере. После этого его следует запустить. Отобразится окно просмотра сертификата (Рисунок 1).

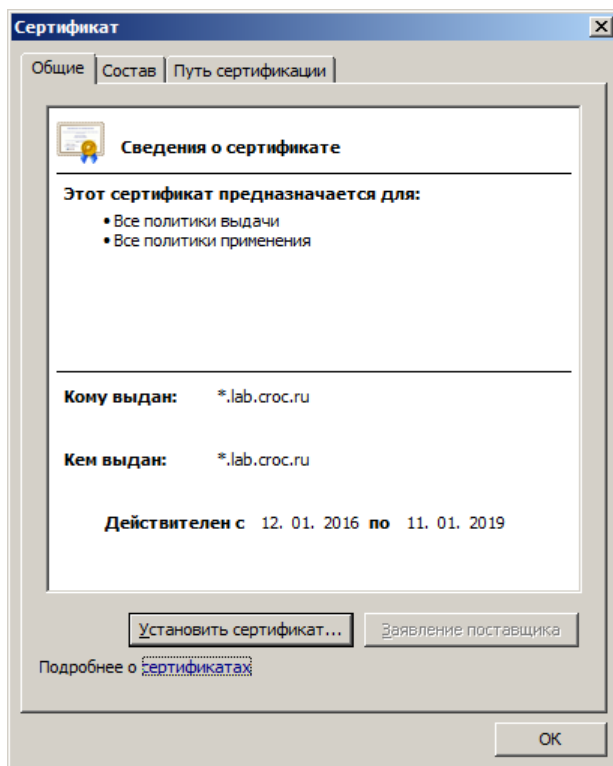


Рисунок 1. Просмотр сертификата

Следует нажать кнопку «Установить сертификат». В первом окне мастера нажать кнопку «Далее» (Рисунок 2).

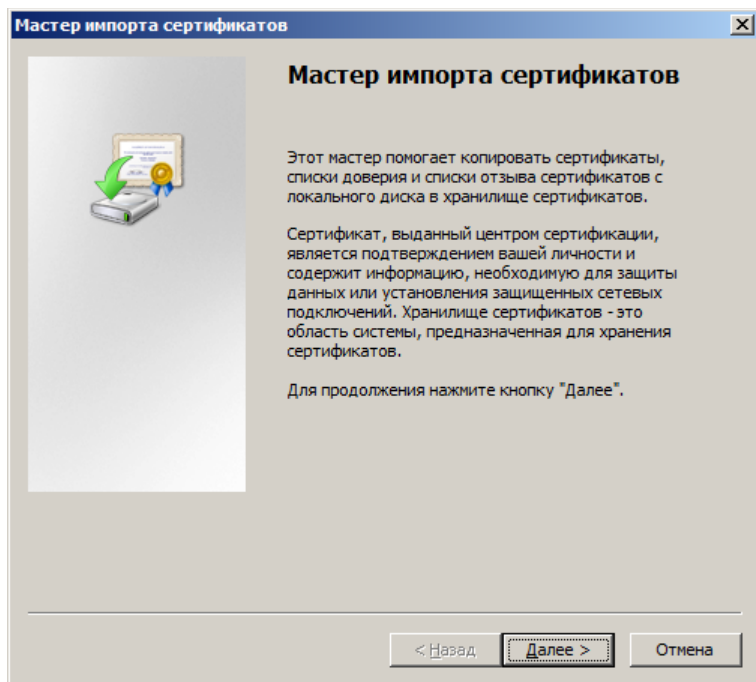


Рисунок 2. Начало процесса импорта сертификата

В появившемся окне выбрать пункт «Поместить все сертификаты в следующее хранилище», нажать кнопку «Обзор» и выбрать раздел «Доверенные корневые центры сертификации» (Рисунок 3).

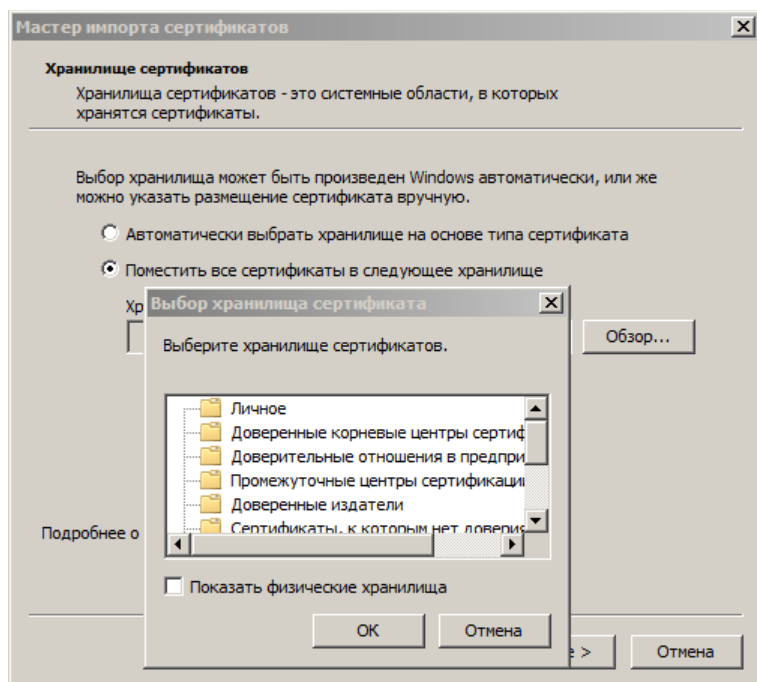


Рисунок 3. Выбор места установки сертификата

В следующем окне следует нажать кнопку «Готово» для завершения импорта (Рисунок 4).

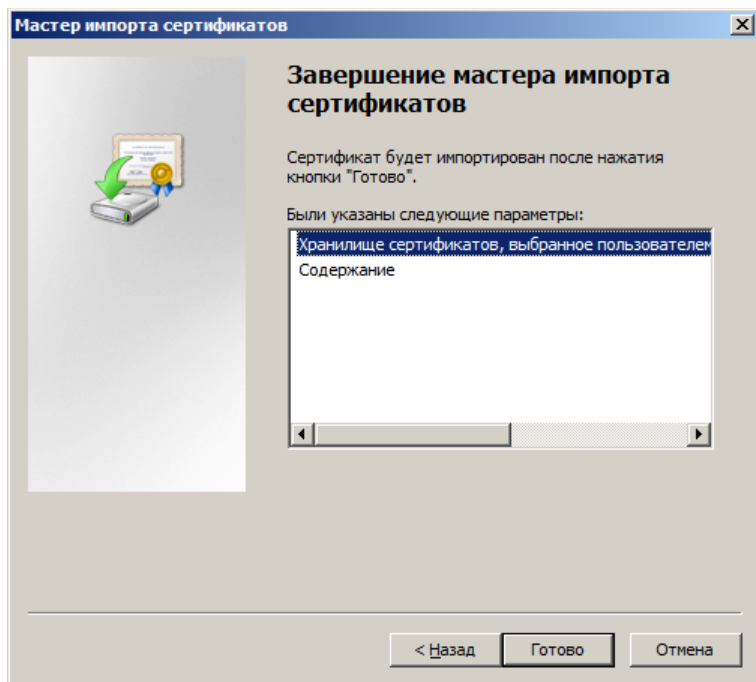


Рисунок 4. Завершение импорта

Так как речь идет о корневом сертификате, выводится дополнительное предупреждение безопасности (Рисунок 5).

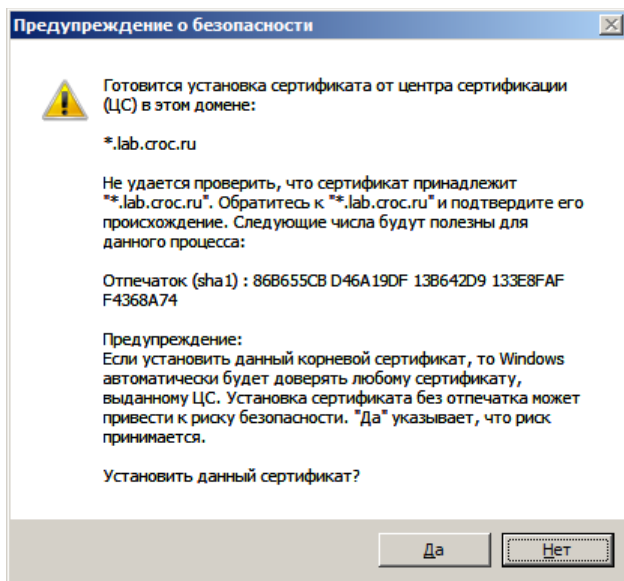


Рисунок 5. Предупреждение о корневом сертификате

После успешного импорта сертификата выводится сообщение (Рисунок 6).

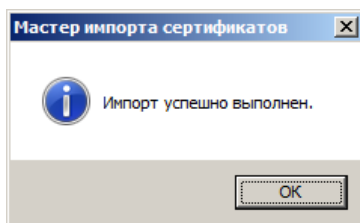


Рисунок 6. Успешное завершение импорта

Для проверки корректности импорта следует открыть портал TFS в браузере по адресу, предоставленному преподавателем, например <https://csharp.lab.croc.ru/tfs>. Если при открытии страницы не выводится предупреждений безопасности, значит, импорт сертификата осуществлен корректно (Рисунок 7).

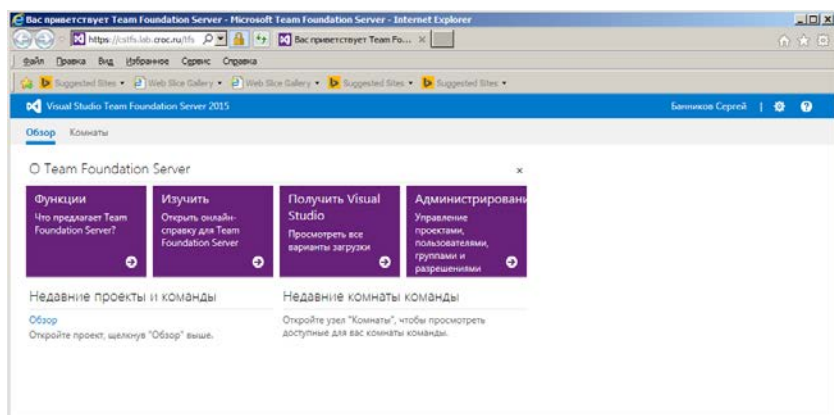



Рисунок 7. Портал Team Foundation Server

1.3. Настройка среды разработки

Порядок подключения к репозитарию исходных текстов на базе Microsoft Team Foundation Server (TFS) следующий:

1. Создать новый пустой каталог, в котором будут размещаться исходные тексты из репозитория TFS. Для определенности пусть это будет каталог C:\CS.
2. Запустить Microsoft Visual Studio 2015
3. Перейти в окно Team Explorer (главное меню View | Team Explorer)
4. Нажать значок  для перехода в режим управления соединениями (Manage Connections)
5. Выбрать пункт Manage Connections | Connect to Team Project (Рисунок 8)

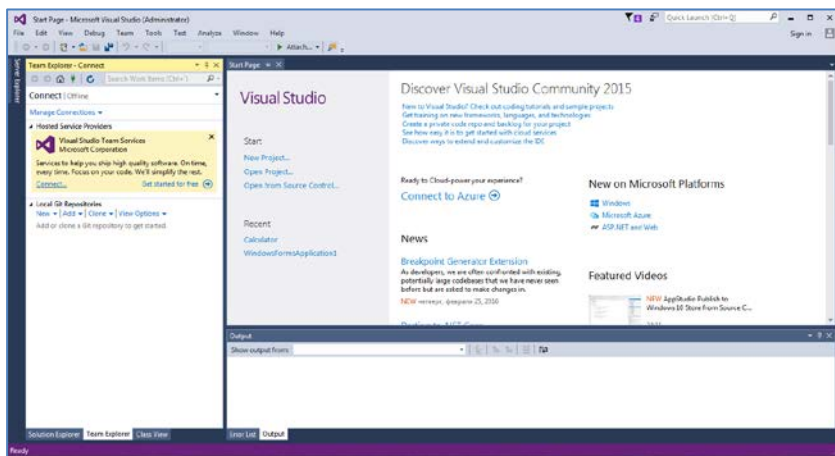


Рисунок 8. Панель Team Explorer

6. В окне Connect to Team Foundation Server нажать кнопку [Servers...] (Рисунок 9)

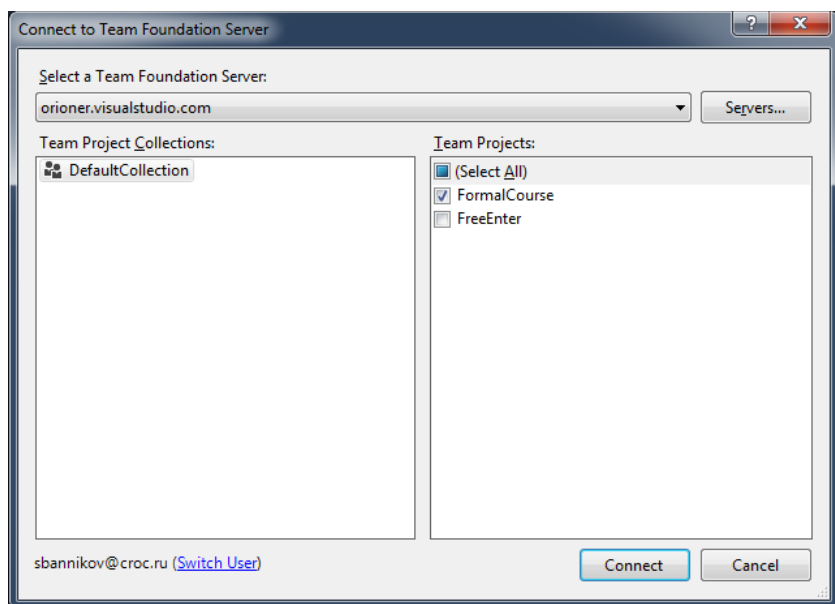


Рисунок 9. Соединение с сервером

7. В окне Add/Remove Team Foundation Server нажать кнопку [Add...] (Рисунок 10)

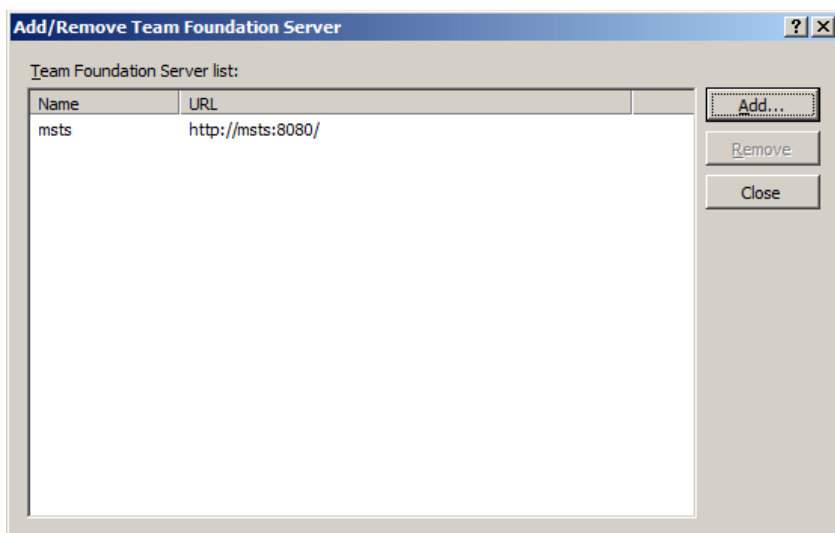


Рисунок 10. Список доступных серверов

8. В окне Add Team Foundation Server ввести имя сервера, выбрать режим подключения (HTTP или HTTPS) и нажать кнопку OK (Рисунок 11). Имя сервера (параметр Name or URL of Team Foundation Server) и режим подключения (Protocol) следует уточнить у преподавателя.

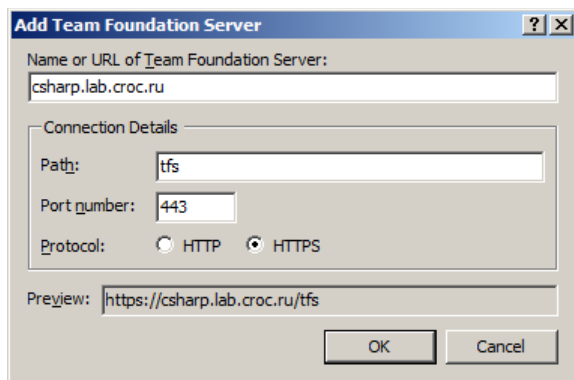


Рисунок 11. Параметры подключения к TFS

9. Далее выводится запрос безопасности Windows (Рисунок 12). Необходимо ввести данные учетной записи в виде SERVER\USERNAME (где SERVER – краткое имя сервера TFS, в данном случае – CSHARP; USERNAME – имя учетной записи студента) и пароль учетной записи, после чего нажать кнопку ОК. Имя учетной записи студента и пароль учетной записи следует получить у преподавателя.

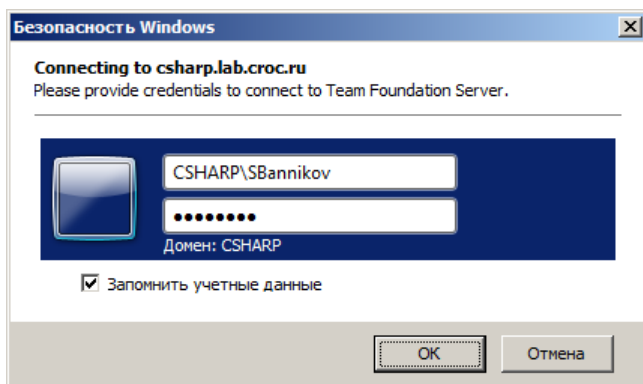


Рисунок 12. Запрос учетных данных

10. Сервер будет добавлен в список серверов. Для того чтобы закрыть окно Add/Remove Team Foundation Server, необходимо нажать кнопку Close (Рисунок 13).

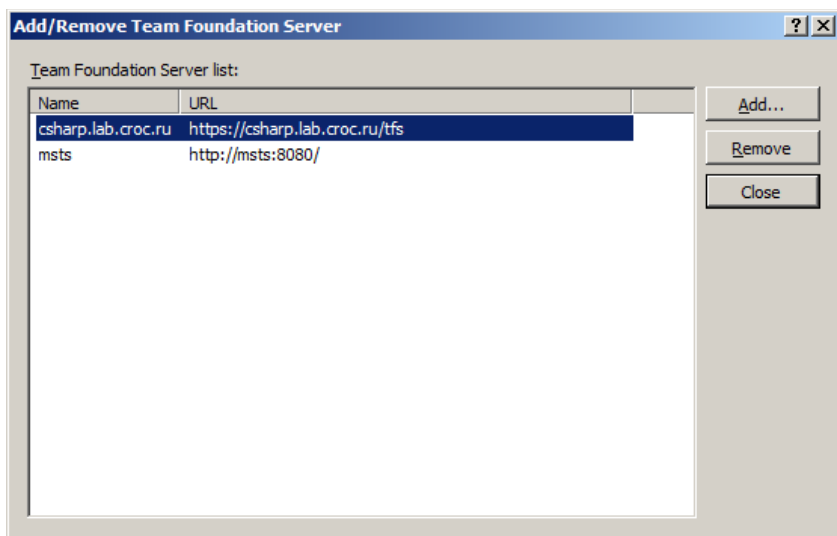


Рисунок 13. Список доступных серверов с добавленным сервером

11. В окне Connect to Team Foundation Server в поле Select a Team Foundation Server следует выбрать вновь добавленный сервер. Выбрать доступную коллекцию проектов (Team Project Collections) – обычно это Default Collection. Выбрать все доступные проекты (Select All). Нажать кнопку Connect (Рисунок 14).

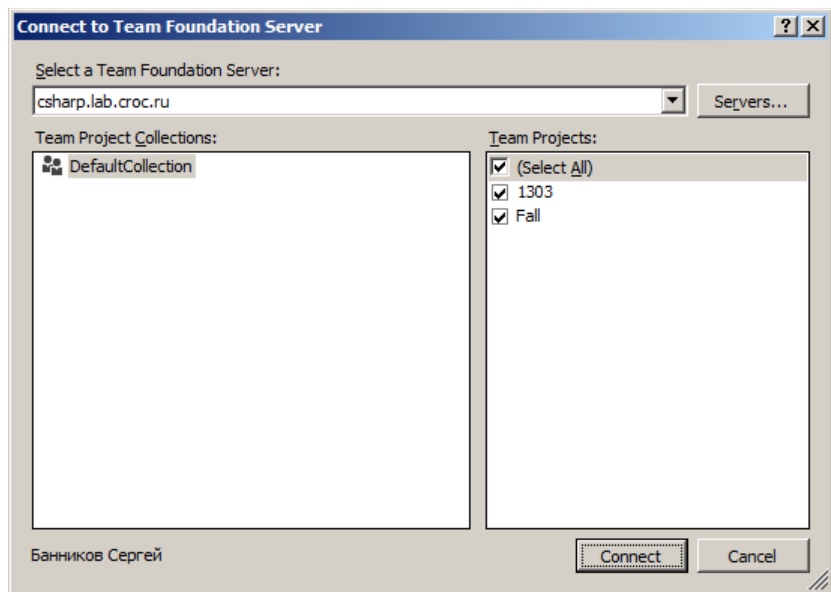


Рисунок 14. Установление соединения с сервером

12. Выбрать окно управления исходными текстами. Для этого используется пункт главного меню View | Other Windows | Source Control Explorer или кнопка Source Control Explorer в окне Team Explorer (Рисунок 15).

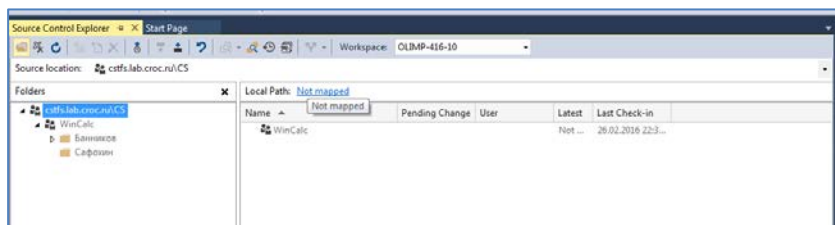


Рисунок 15. Панель Source Control Explorer

13. В левой панели Folders выбрать корневой узел. В правой панели нажать на ссылку Not Mapped (Рисунок 15).

14. В окне Map выбрать локальный каталог, созданный выше. Нажать кнопку Map (Рисунок 16). В указанный каталог будет осуществлена загрузка всех исходных текстов.

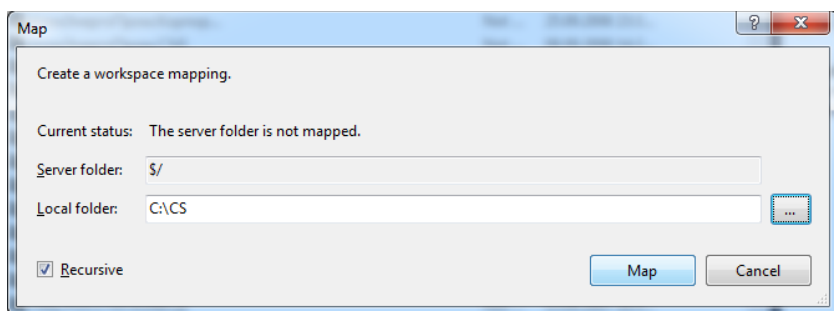


Рисунок 16. Отображение репозитория на локальную папку

15. Для открытия проекта (project) или решения (solution) следует выбрать соответствующий файл в окне Source Control Explorer при помощи двойного щелчка мыши (Рисунок 17).

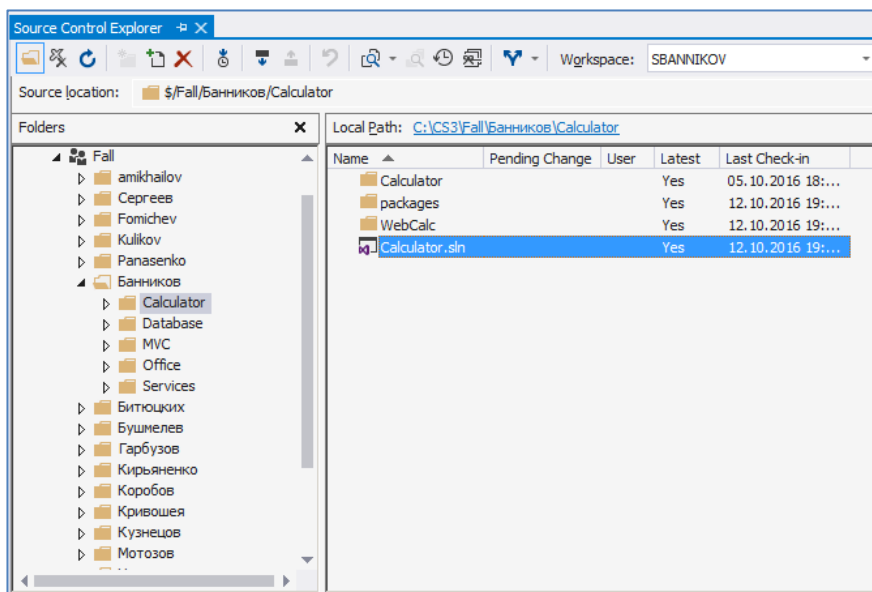


Рисунок 17. Окно Source Control Explorer

1.4. Учетные данные при подключении к TFS

При подключении к TFS учетные данные (имя пользователя и пароль) сохраняются в диспетчере учетных записей. Если необходимо подключиться к репозитарию от имени другого пользователя (например, при работе в классе), то необходимо удалить сохраненную учетную запись из диспетчера учетных записей. Для этого необходимо открыть панель управления. Далее следует выбрать пункт «Учетные записи пользователей» (Рисунок 18).

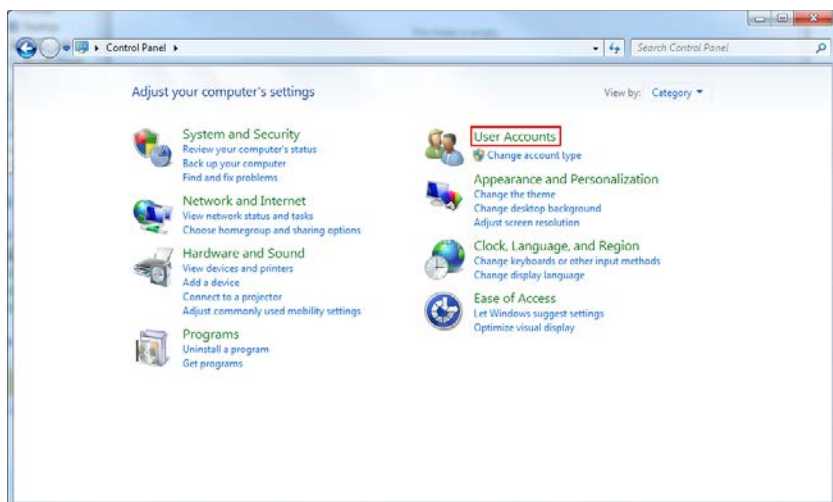
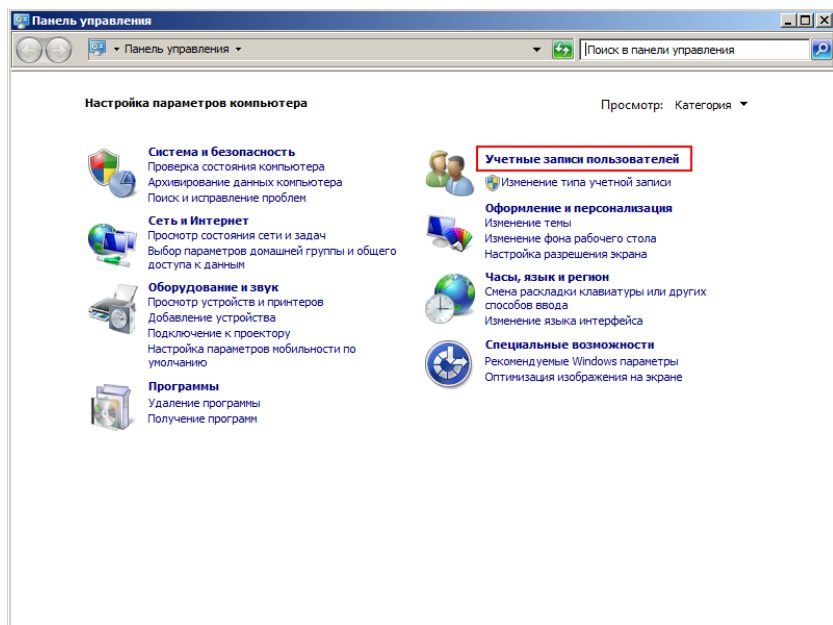


Рисунок 18. Панель управления

В окне учетных записей пользователей следует выбрать пункт «Диспетчер учетных данных» (Рисунок 19).

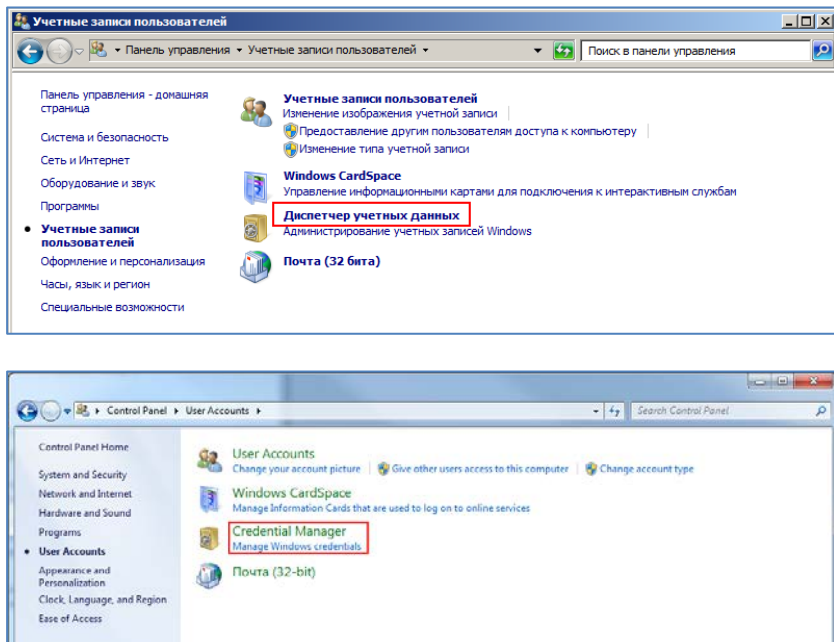


Рисунок 19. Учетные записи пользователей

В окне диспетчера учетных данных следует найти подключение к серверу TFS (csharp.lab.croc.ru). Следует обратить внимание, что таких подключений может быть несколько (Рисунок 20).

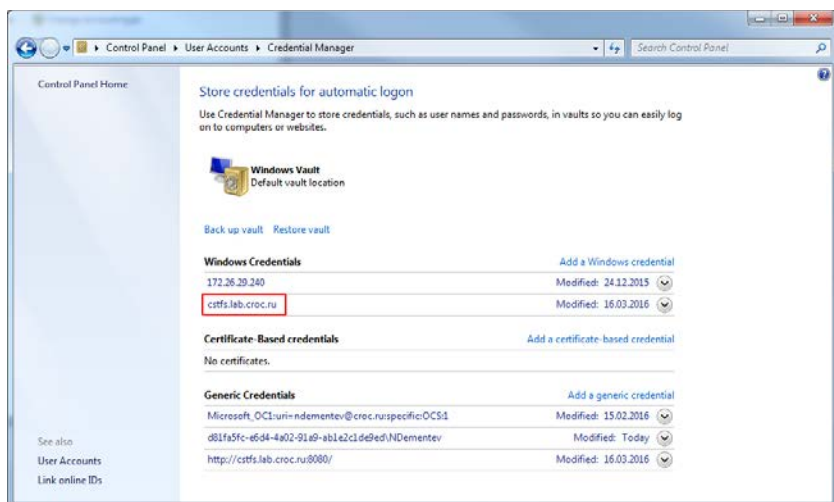
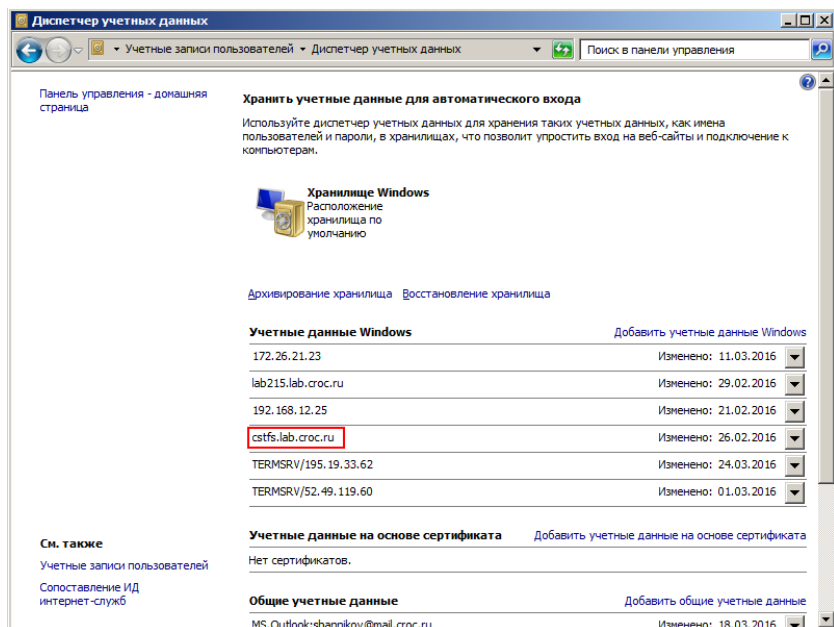


Рисунок 20. Диспетчер учетных записей

Для каждого найденного подключения следует выбрать пункт «Удаление из хранилища». В появившемся предупреждающем окне следует согласиться с удалением (Рисунок 21). Данную операцию следует повторить для всех учетных данных, относящихся к репозиторию TFS – серверу `csharp.lab.croc.ru`.

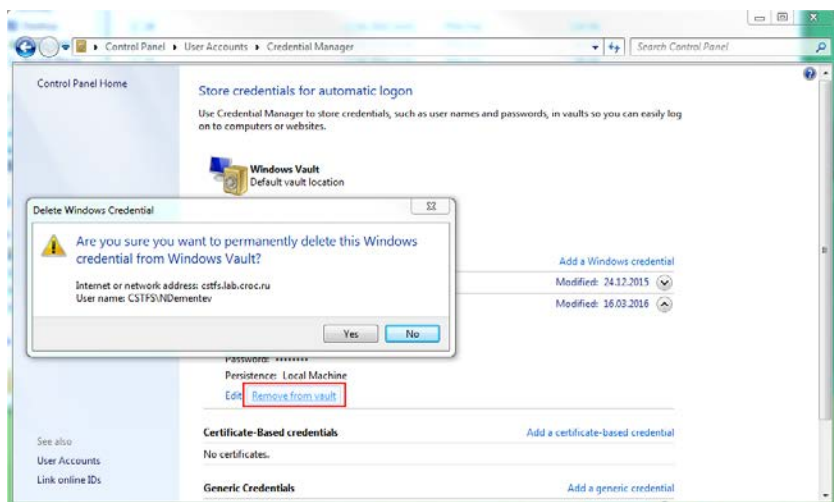
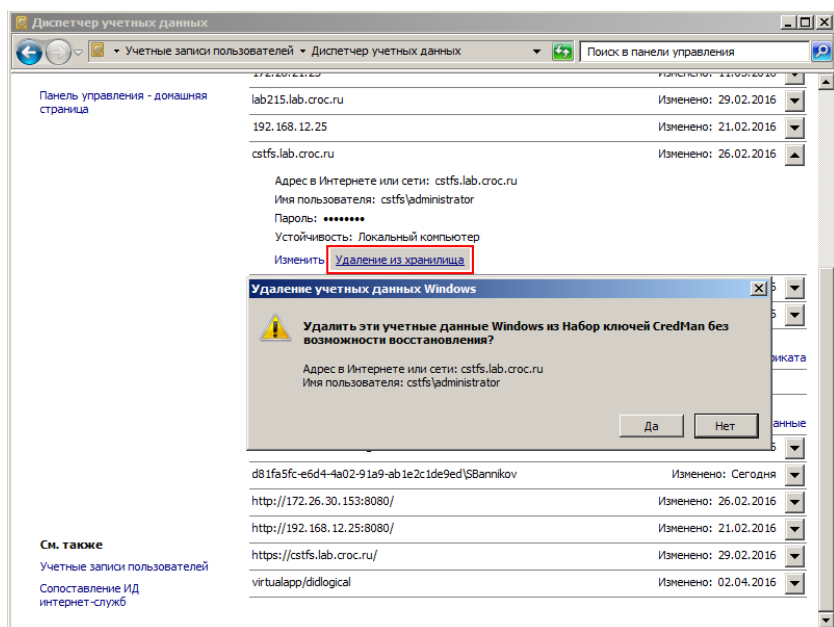


Рисунок 21. Удаление сохраненных данных

1.5. Использование

1. Для того чтобы поместить уже существующий проект в репозиторий, необходимо скопировать его в свою папку (в соответствии с фамилией студента). Далее следует в окне Solution Explorer выделить файл решения и в контекстном меню по правой клавише мыши выбрать пункт Add Solution To Source Control (Рисунок 22).

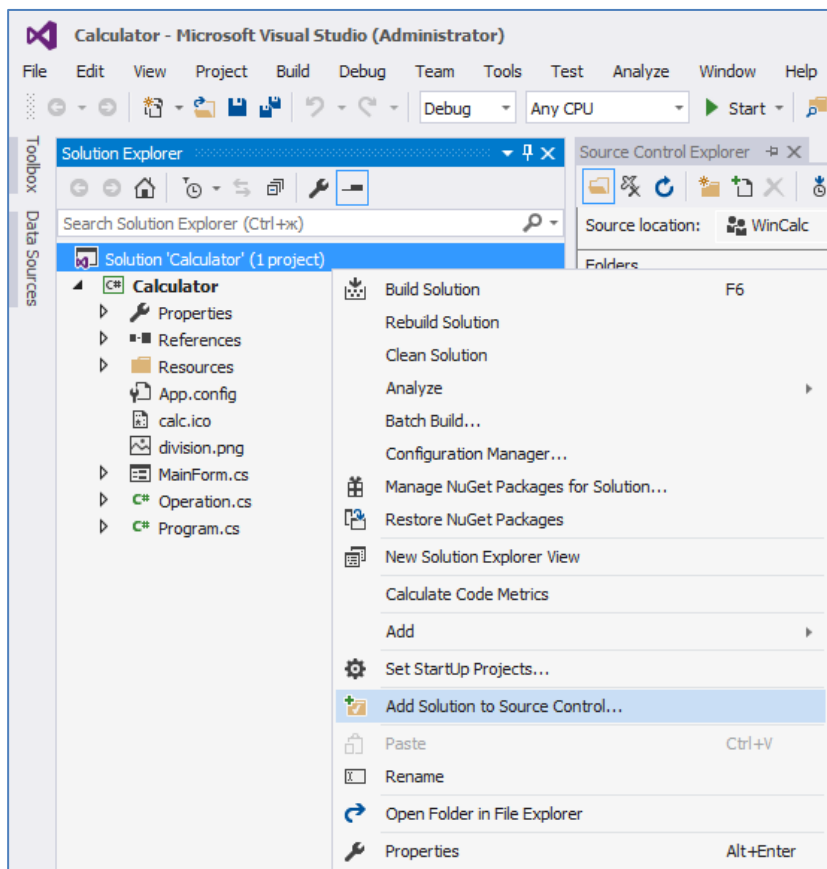


Рисунок 22. Добавление проекта в репозиторий

2. Для записи файлов в репозиторий следует в окне Team Explorer переключиться в режим Pending Changes (если это не произошло автоматически), заполнить поле комментария (Comment) и нажать кнопку Check In (Рисунок 23).

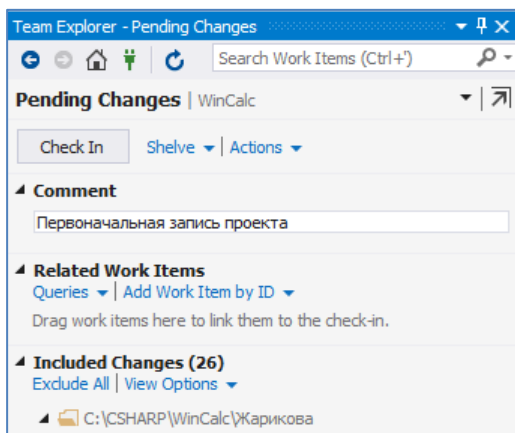


Рисунок 23. Запись изменений в репозиторий

3. При редактировании любого файла его выписка из репозитория (Check Out) происходит автоматически. После внесения изменений следует выполнить запись (Check In) так же, как и при первоначальной записи.
4. Отмена сделанных изменений и возврат к предыдущей сохраненной в репозитории версии осуществляется в окне Source Control Explorer при помощи пункта контекстного меню Undo Pending Changes.
5. Сравнение различных версий файлов осуществляется в окне Source Control Explorer при помощи пункта контекстного меню Compare.

1.6. Возможные проблемы при сборке проекта

В данном разделе содержится описание типовых проблем, которые могут возникать при сборке проекта.

1.6.1. Системная сборка не может быть загружена

При сборке проекта выводится сообщение

The "Microsoft.CodeAnalysis.BuildTasks.Csc" task could not be loaded from the assembly
xxx\packages\Microsoft.Net.Compilers.1.0.0\build\..\tools\Microsoft.Build.Tasks.CodeAnalysis.dll. Could not load file or assembly
'xxx\packages\Microsoft.Net.Compilers.1.0.0\tools\Microsoft.Build.Tasks.CodeAnalysis.dll' or one of its dependencies. Не удается найти указанный файл. Confirm that the <UsingTask> declaration is correct, that the assembly and all its dependencies are available, and that the task contains a public class that implements Microsoft.Build.Framework.ITask.

Проблема в том, что в репозиторий TFS оказались записаны не только исходные коды приложения, но и пакеты NuGet, как показано ниже (Рисунок 24).

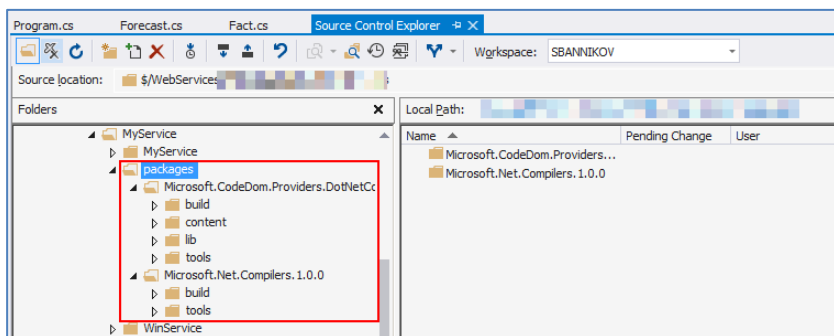


Рисунок 24. Пакеты NuGet, записанные в репозиторий

Пакеты NuGet загружаются при сборке проекта автоматически, и их записывать в репозиторий TFS не требуется. Если же по какой-то

причине пакеты не были загружены автоматически, то при помощи команды **Restore NuGet Packages** в контекстном меню по правой клавише мыши на файле решения (solution) пакеты будут загружены принудительно (Рисунок 25).

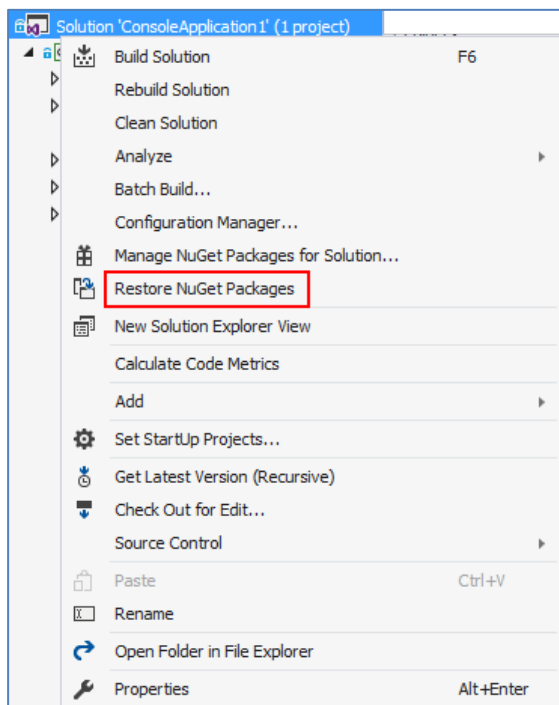


Рисунок 25. Восстановление пакетов NuGet

Для устранения данной ошибки следует удалить из TFS папку `packages`.

2. Разработка простого калькулятора на базе WinForms

2.1. Создание проекта

Создание любого приложения в интегрированной среде разработки Microsoft Visual Studio начинается с создания нового проекта. Для этого используется пункт главного меню «File | New | Project» (Рисунок 26). В появившемся окне следует выбрать тип проекта, корневую папку для его размещения, задать имя проекта и имя решения. Для добавления проекта в систему управления исходными текстами необходимо включить режим «Add to Source Control». Для создания отдельного каталога для проекта внутри решения крайне рекомендуется включить также режим «Create directory for solution». Версию .NET Framework для данного задания можно выбрать произвольную, но рекомендуется выбирать версию не ниже 4.5.

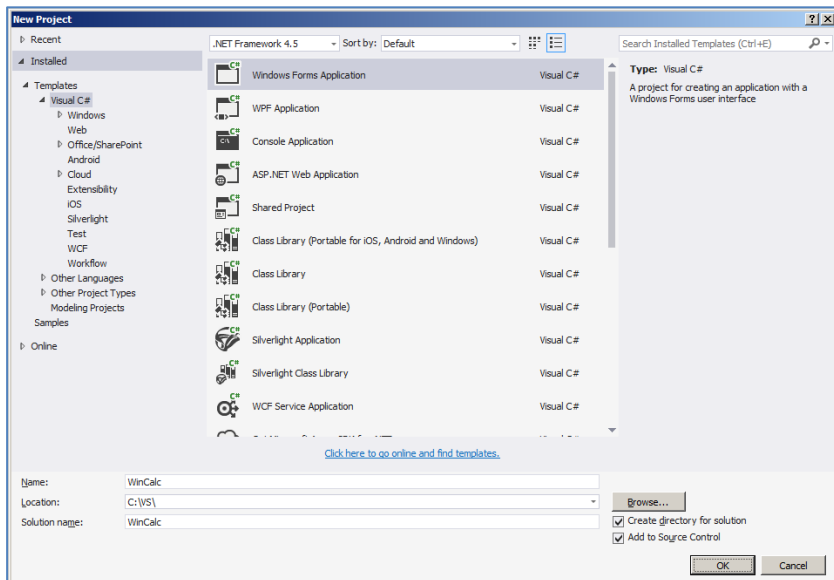


Рисунок 26. Создание нового проекта

После того, как проект успешно создан, имеет смысл сразу записать его в репозиторий исходных текстов. Для этого в окне Solution Explorer (Рисунок 27) на файле решения (Solution) необходимо нажать правую клавишу мыши и в контекстном меню выбрать пункт «Check-in». При этом стоит обратить внимание, что все файлы проекта помечены значком «+», что означает, что они пока существуют только локально, на компьютере разработчика, и еще не записаны в репозиторий.

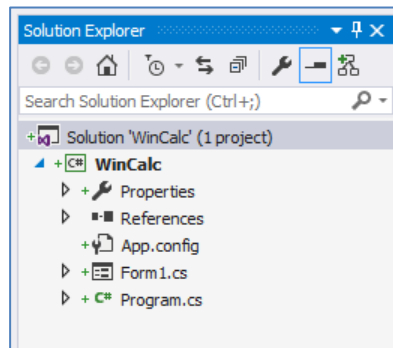


Рисунок 27. Вновь созданный проект

Окно переключается в режим записи изменений (Рисунок 28). Следует внести комментарий и нажать кнопку «Check In».

В промышленной разработке программного обеспечения обычно используют политику, предотвращающую запись изменений исходных текстов без комментариев. Если такая политика активирована, при попытке записи изменений без комментариев выводится сообщение «Check-in validation failed. A policy warning override reason and/or a check-in note is required», и запись изменений становится невозможной. Единственным способом записи без комментария является предоставление комментария, почему не может быть за-

полнен комментарий (override reason) – так или иначе, но политика обязывает разработчика комментировать собственные действия.

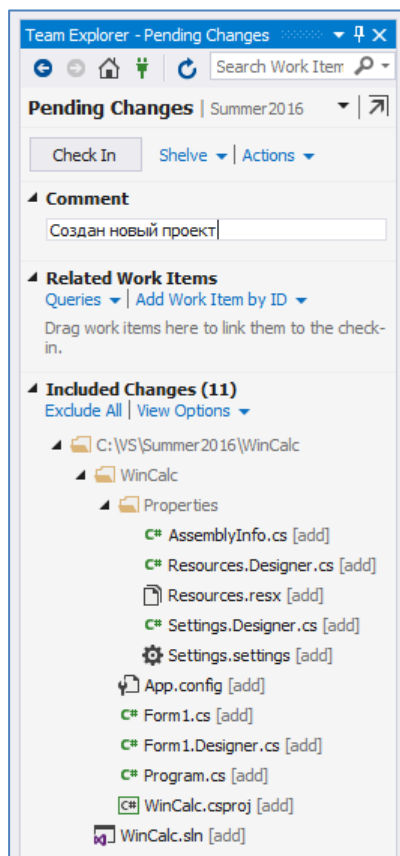


Рисунок 28. Добавление проекта в репозиторий исходных текстов

После нажатия кнопки «Check In» выводится дополнительный запрос подтверждения. Рекомендуется отключить его на будущее. Впрочем, если нет уверенности в собственных действиях, то данное предупреждение может оказаться полезным.

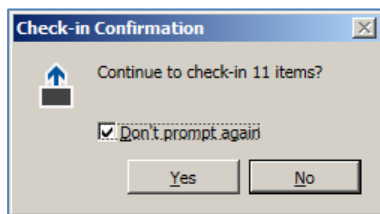


Рисунок 29. Подтверждение записи изменений

Для запуска вновь созданного приложения используется кнопка «Start», пункт главного меню «Debug | Start Debugging» или клавиша F5. Так как созданное приложение пока не содержит никакого полезного кода, в результате запуска получается пустая форма. Но так как приложение наследует все свойства стандартной формы (класс `System.Windows.Forms.Form`), то созданная пустая форма уже реализует все стандартные операции формы по умолчанию – минимизация формы (сворачивание в значок), максимизация формы (распаивание на весь экран), закрытие формы (завершение работы приложения), изменение размера формы и ее перемещение по экрану.

2.2. Свойства проекта

Для просмотра и редактирования свойств любого проекта в интегрированной среде разработки Microsoft Visual Studio необходимо в панели Solution Explorer дважды щелкнуть по пункту Properties (Рисунок 30).

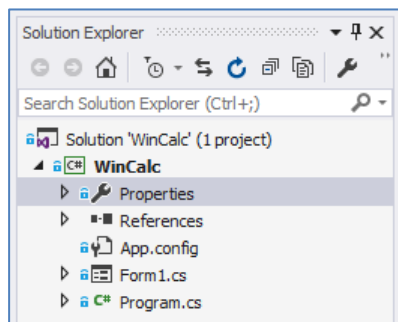


Рисунок 30. Свойства проекта в иерархии решения

Появившееся окно (Рисунок 31) имеет несколько разделов (состав разделов меняется в зависимости от типа приложения), наиболее важные из которых описаны ниже.

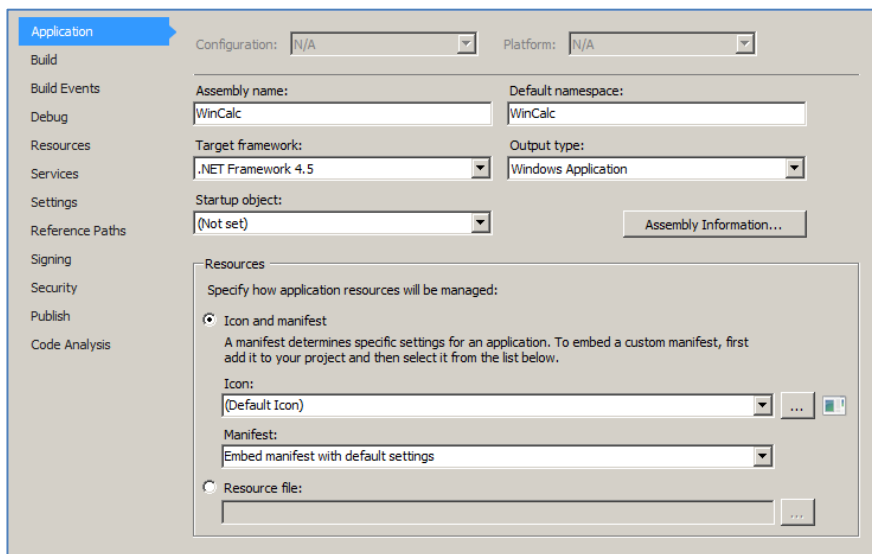


Рисунок 31. Окно свойств проекта

2.2.1. Свойства приложения

Раздел «Application» (Рисунок 31) содержит основные настройки проекта. Наиболее важными являются следующие поля:

Assembly name – имя исполняемого файла проекта (сборки). Расширение файла (EXE или DLL) зависит от типа проекта (поля **Output type**). Всего существуют три базовых типа проектов на C#:

- **Windows Application** – приложение с графическим интерфейсом, расширение файла EXE (сокращение от executable).
- **Console Application** – приложение с интерфейсом командной строки, расширение файла EXE.
- **Class library** – динамическая библиотека классов, расширение файла DLL (dynamic link library).

В данном случае, так как создается приложение с графическим интерфейсом, тип проекта должен быть Windows Application.

Target framework – версия библиотеки Microsoft .NET Framework, начиная от 2.0 и заканчивая последней доступной версией (на текущий момент это версия 4.6.1). От версии библиотеки зависит доступный набор системных классов и возможностей, поэтому имеет смысл использовать последнюю доступную версию. В то же время самые последние версии могут быть недоступны на устаревающих версиях операционных систем, поэтому если разрабатываемое приложение предназначено, скажем, для использования в Windows XP, следует уточнить, какую версию .NET Framework имеет смысл использовать. Следует также знать, что для версий с 2.0 по 3.5 используется среда исполнения CLR (Common Language Runtime) версии 2.0, а начиная с версии 4.0, используется среда исполнения версии 4.0. На это требуется обращать отдельное внимание в тех случаях, когда, например, разрабатывается веб-приложение для использо-

вания в среде IIS (Internet Information Services – встроенный веб-сервер операционной системы Windows). Для корректного запуска такого приложения IIS должен быть настроен соответствующим образом.

Default namespace – пространство имен по умолчанию. В языке C# все идентификаторы существуют в одном общем пространстве именования. Для того чтобы можно было использовать одноименные идентификаторы для обозначения разных объектов, используется механизм пространств имен. Таким образом, например, объект Timer в пространстве имен System.Timers – это совсем не одно и то же, что объект Timer в пространстве имен System.Windows.Forms (Рисунок 32). Пространства имен образуют иерархию, отдельные уровни иерархии отделяются точками. Таким образом, пространство имен System.Windows входит в пространство имен System, а, в свою очередь, пространство имен System.Windows.Forms входит в пространство имен System.Windows. Как правило, каждый проект имеет свое собственное пространство имен.

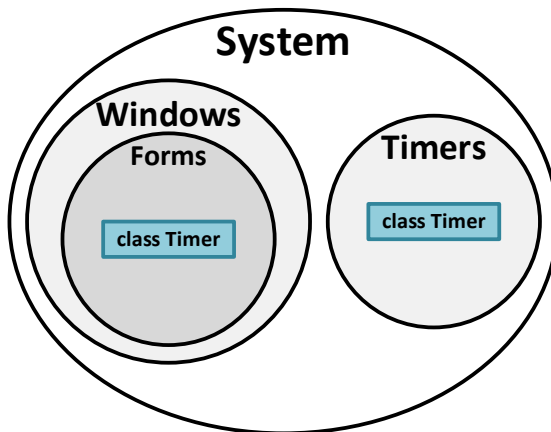


Рисунок 32. Пример структуры пространств имен

Startup object – объект, с которого начинается выполнение программы. По умолчанию это статический класс Program, который, впрочем, можно выбрать и вручную. Если в программе несколько стартовых объектов, можно выбрать один из них. Практического значения это не имеет, так как стартовый объект все равно может быть только один, и определяется это на стадии компиляции программы, а не на стадии ее выполнения.

Icon – значок файла приложения. По умолчанию используется системный значок (Рисунок 33). Но крайне рекомендуется для каждого приложения предусмотреть свой (желательно уникальный) значок.

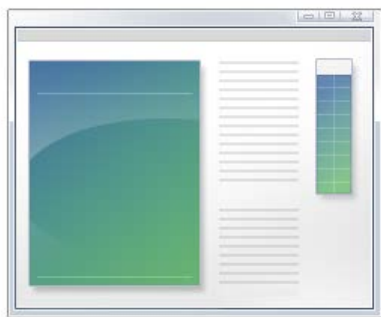


Рисунок 33. Значок файла приложения по умолчанию

Для того чтобы изменить значок приложения, следует нарисовать его в графическом редакторе и сохранить в формате ICO (или найти в Интернет на специализированных сайтах, например, <http://www.iconfinder.com> или <http://www.flaticon.com>). Файл надо скопировать в тот же каталог, в котором размещены исходные тексты проекта. Для добавления файла к проекту следует в панели Solution Explorer выделить файл проекта (Рисунок 34) и в контекстном меню по правой клавише мыши выбрать пункт «Add | Existing item». Далее следует выбрать файл значка и нажать кнопку «Add». Если

система контроля исходных текстов активна, значок файла в дереве будет помечен плюсиком. После добавления файла значка к проекту его можно выбрать в выпадающем списке поля Icon. Альтернативным способом добавления значка является использование кнопки «[...]», которая позволяет выбрать произвольный файл значка в любом месте файловой системы. После выбора файл значка будет скопирован в каталог проекта.

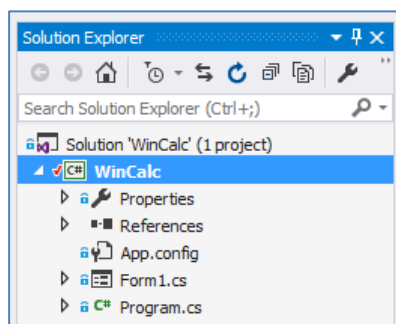


Рисунок 34. Файл проекта в иерархии решения

Примечание. Файл формата ICO может содержать несколько растровых изображений разного размера, что позволяет использовать один и тот же значок при различных режимах просмотра и для разных экранных разрешений (Рисунок 35).

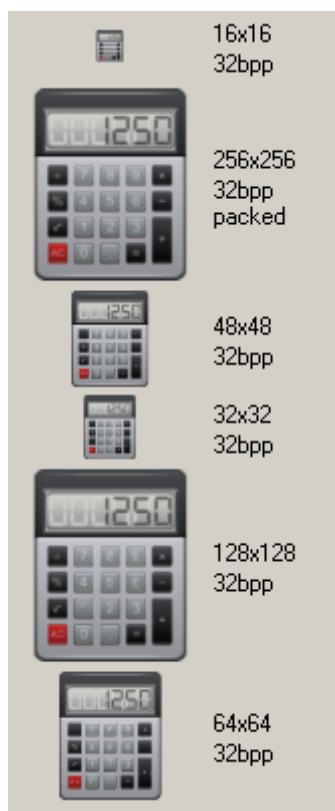


Рисунок 35. Варианты значка для различных размеров

Кнопка **Assembly Information** открывает отдельное окно редактирования информации о файле сборки (Рисунок 36). Эта информация хранится в отдельном файле `AssemblyInfo.cs` в каталоге проекта. При компиляции приложения эта информация становится доступной при просмотре свойств файла из Проводника Windows (Рисунок 37).

Assembly Information [?] [X]

Title: Калькулятор

Description: Простой калькулятор на базе Windows Forms

Company: ЗАО 'КРОК инкорпорейтед'

Product: Учебный курс С#

Copyright: Copyright © 2016 CROC incorporated

Trademark:

Assembly version: 1 0 0 0

File version: 1 0 0 0

GUID: 10444912-e22a-4d61-9e4d-4eb8c6a3c4e1

Neutral language: (None) ▼

☐ Make assembly COM-Visible

OK Cancel

Рисунок 36. Информация о файле сборки

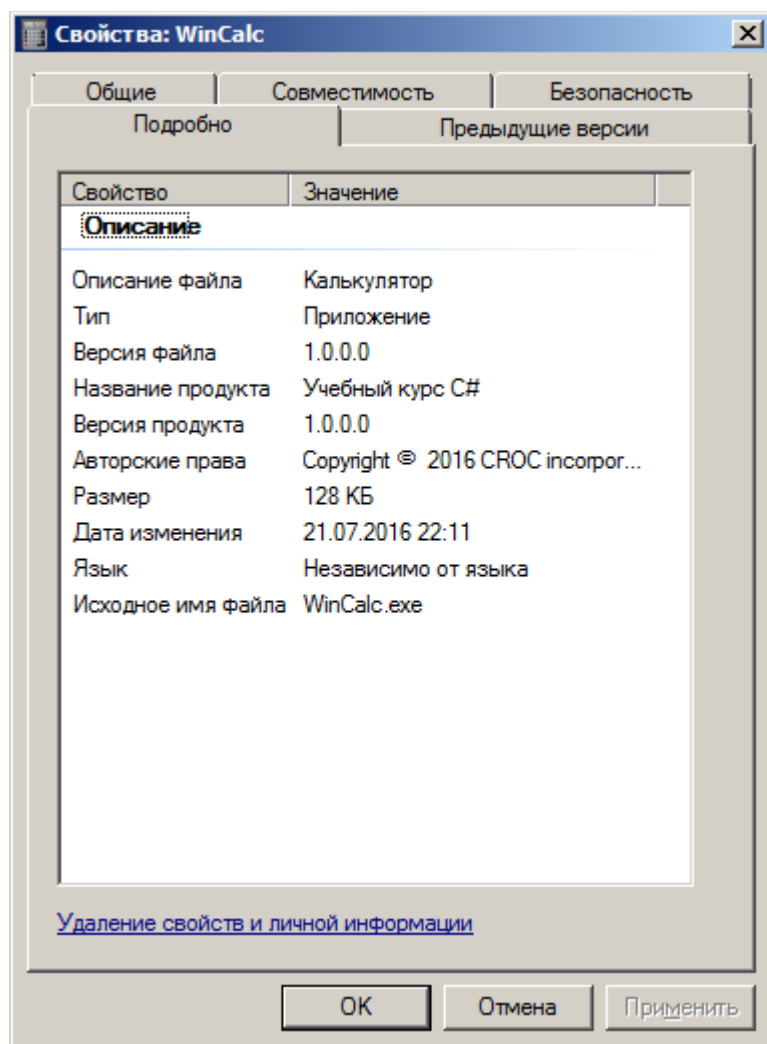


Рисунок 37. Свойства исполняемого файла в Проводнике Windows

Крайне рекомендуется для каждого разрабатываемого проекта заполнять информацию о файле, в частности, номер версии файла (File version). Номер версии файла состоит из четырех чисел, разделен-

ных точками по схеме major.minor.build.private. Традиционное значение этих полей следующее:

- major – главный номер версии
- minor – второстепенный номер версии
- build – номер сборки
- private – внутренний номер

На самом деле каждый понимает номер версии в меру собственных представлений о том, как надо. Но при сколь-либо длительной разработке следует выработать политику изменения номеров версий файла. Например, может быть предложена следующая политика:

1. Нумерация версии начинается с 1.0.0.0.
2. После каждого изменения программы и копирования ее с машины разработчика в тестовую среду (то есть когда разработчик отпускает свой продукт в мир) внутренний номер увеличивается на единицу. Например, от версии 1.0.0.0 мы переходим к 1.0.0.1.
3. После реализации очередного требования номер сборки увеличивается на единицу. Внутренний номер при этом может быть сброшен в ноль или также увеличен (в соответствии с предыдущим правилом): 1.0.1.0 или 1.0.1.2
4. После реализации очередного связанного набора требований второстепенный номер увеличивается на единицу, номер сборки сбрасывается в 0, внутренний номер при этом может быть сброшен в ноль или увеличен: 1.1.0.0 или 1.1.0.3.
5. После завершения существенного этапа проекта или проекта целиком (или при начале следующего этапа или проекта) главный номер увеличивается на единицу, второстепенный номер и номер сборки сбрасываются в 0, внутренний номер

при этом может быть сброшен в ноль или увеличен: 2.0.0.0 или 2.0.0.4.

Непрерывное увеличение внутреннего номера хранит историю об эволюции программы с самого начала. Так, в одном из проектов, выполненных автором, номер версии файла дошел до 1.17.0.615.

Внимание! Не следует путать номер версии сборки (Assembly version) и номер версии файла (File version). Номер версии файла имеет описательное значение (кроме того, он может использоваться инсталляторами для того, чтобы различать версии файлов в процессе установки). Номер версии сборки используется для того, чтобы различить различные программные варианты реализации программы друг от друга, при этом в одной системе могут присутствовать одноименные файлы, но имеющие различные номера версий сборки. Сложное приложение, состоящее из нескольких файлов, ссылается на вспомогательные сборки, используя не только имя сборки, но и номер версии сборки. Это позволяет различным версиям одной и той же программы относительно мирно сосуществовать на одном компьютере.

Кроме описательной информации, в сведениях о файле сборки имеются три важных поля, описанные ниже.

GUID – уникальный идентификатор проекта. Формируется при создании проекта автоматически. В исключительно редких случаях требуется задавать его вручную (например, если с нуля создается новый вариант старой библиотеки).

Neutral language – указывает язык (языковой ресурс), который используется по умолчанию в тех случаях, когда языковые ресурсы не могут быть загружены для языка операционной системы, где запущена программа. С этим параметром имеет смысл разбираться при

разработке многоязычных приложений, предназначенных для работы в различных языках. По умолчанию никакой язык не выбран.

Make assembly COM-visible – разрешает обращение к программному коду при помощи технологии COM. Требуется всё реже и реже, но при разработке встраиваемых модулей для устаревающих приложений может оказаться необходимым. По умолчанию режим отключен.

Файл информации о сборке AssemblyInfo.cs расположен в иерархии проекта под его свойствами и по умолчанию не виден (Рисунок 38). Обычно его не редактируют вручную, но это тоже возможно. В качестве упражнения предлагается открыть и ознакомиться с содержанием данного файла самостоятельно.

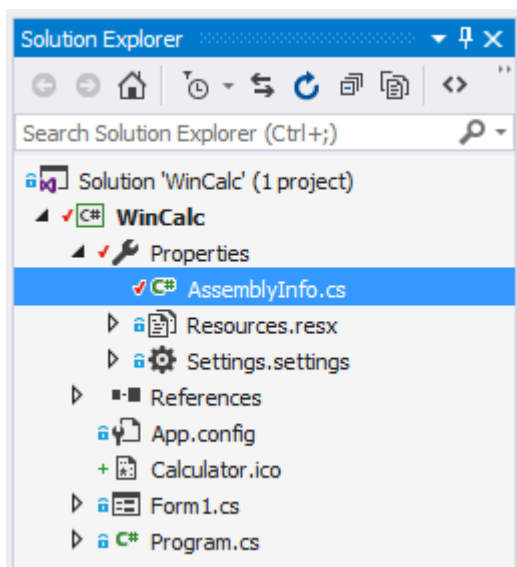


Рисунок 38. Расположение файла AssemblyInfo.cs в иерархии проекта

2.2.2. Остальные разделы свойств проекта

Окно «Свойства проекта» содержит и другие разделы, перечисленные ниже (Таблица 1). Для целей данного занятия изменять их не требуется.

Таблица 1. Перечень разделов

Закладка	Назначение
Application	Свойства приложения (рассмотрены в разделе 2.2.1 выше)
Build	Настройки сборки проекта
Build Events	Описание дополнительных действий, которые могут осуществляться до или после сборки проекта
Debug	Настройка отладки приложения
Resources	Ресурсы проекта – строки, изображения, значки, звуки, файлы.
Services	Настройки прикладных сервисов, используемых для авторизации, построения ролевой модели, работы с профилями пользователей
Settings	Настройки приложения, сохраняются в отдельном конфигурационном файле *.config
Reference Paths	Перечень дополнительных путей для поиска используемых библиотек
Signing	Добавление цифровой подписи к приложению
Security	Настройки безопасности для приложений, использующих технологию ClickOnce
Publish	Публикация веб-приложений
Code Analysis	Настройки анализа кода

2.3. Главная функция программы

Главная функция программы всегда носит название `Main()` и по умолчанию располагается в классе `Program` в файле `Program.cs`. Сразу после создания проекта она выглядит следующим образом (Рисунок 39).

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WinCalc
{
    static class Program
    {
        /// <summary>
        /// Главная функция приложения
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}

```

Рисунок 39. Главная функция программы

Разберем ключевые слова и конструкции языка по порядку их использования в коде (Рисунок 39).

using – в данном случае это директива, которая используется для подключения пространства имен и соответствующей библиотеки. Так как одно и то же пространство имен может быть использовано в разных библиотеках, а имя файла может не совпадать с именем пространства имен, не следует думать, что, например, директива `using System` подключает к проекту только файл `System.dll`. Список подключенных файлов можно увидеть в иерархии проекта в разделе `References` (Рисунок 40), и, как легко увидеть, он отличается от списка пространств имен, которые подключаются при помощи директив `using` в нашей программе.

namespace – объявление пространства имен. Все, что внутри блока { ... }, будет относиться к пространству имен WinCalc. Для обращения к объектам из других пространств имен обычно требуется использовать полный идентификатор объекта, включающий имя пространства и имя объекта через точку. Но так как выше использована директива using, для пространства имен System и прочих указанных в директивах, это не требуется, и класс Sytem.Windows.Forms.Application указан просто как Application.

static – применительно к классу это означает, что данный класс будет содержать только статические методы, то есть методы, описанные как static. Ниже, применительно к методу Main() это означает, что данный метод вызывается без контекста экземпляра класса (объекта). Фактически – это просто функция в терминах языка программирования C.

class – с этого ключевого слова начинается описание класса. Program – это его имя.

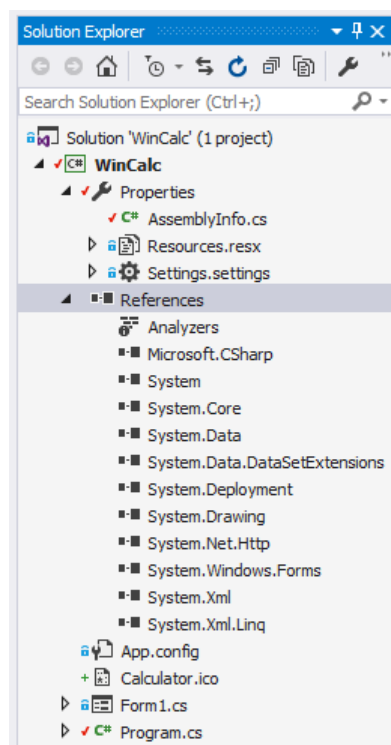


Рисунок 40. Список внешних ссылок проекта

void — означает, что функция не возвращает результата своего выполнения. В терминах языка Паскаль такая функция называлась бы процедурой.

Application — системный класс, реализующий приложение. Наше приложение состоит из одной формы **Form1**. Для запуска формы на выполнение используется статический метод **Run**. Статический он потому, что мы не создаем объекта приложения в явной форме.

Зато мы создает объект класса **Form1** при помощи ключевого слова **new**. Ключевое слово **new** используется для создания новых объек-

тов. Будучи созданной, форма запускается при помощи метода Run, и мы видим ее на экране.

Выделенные желтым (Рисунок 39) конструкции пока не совсем понятны. Они могут быть пока просто удалены. Когда (и если) их потребуется добавить, это будет сделано осознанно. После изменения исходных текстов следует проверить, что приложение работает, и записать результат в репозиторий при помощи команды Checkin.

2.4. Главная форма

Главная форма приложения по умолчанию называется Form1. Это не совсем приличное название для главной формы. Рекомендуется называть ее MainForm, тогда никто не будет оспаривать того факта, что она главная. Для переименования формы следует выделить ее в иерархии проекта в окне Solution Explorer и в контекстном меню по правой клавише мыши выбрать пункт Rename. Так как мы переименовываем файл, содержащий класс (а форма – это класс), то выводится следующее сообщение (Рисунок 41).

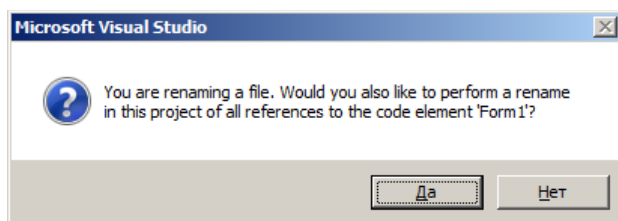


Рисунок 41. Переименование файла

Следует обязательно согласиться с автоматическим переименованием класса и всех ссылок на него, иначе эту совершенно неблагодарную работу придется делать вручную. После этого следует обратить внимание на то, что теперь запуск формы в главной функции программы выглядит как `Application.Run(new MainForm())`.

Для того, чтобы открыть главную форму, требуется два раза щелкнуть на соответствующем имени файле в окне Solution Explorer. На экране появится так называемый дизайнер – окно для визуального редактирования формы. Рядом отображаются свойства (Properties) выбранного компонента. Так как ни одного компонента еще не выбрано, отображаются свойства формы. Если свойства не отображаются, необходимо нажать клавишу F4 или выбрать пункт главного меню «View | Property Window» (Рисунок 42).

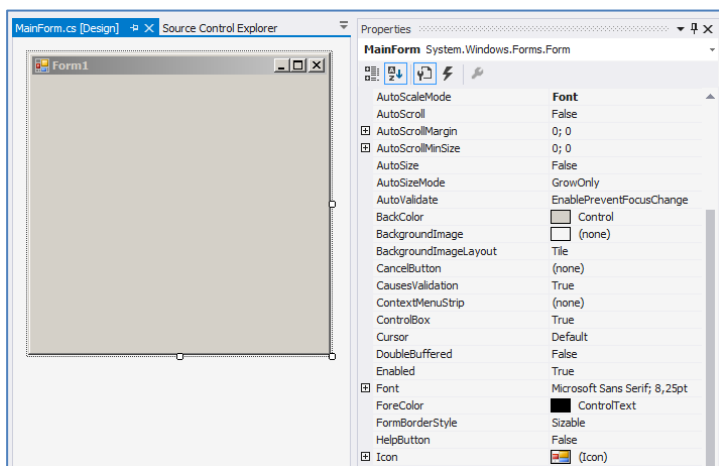


Рисунок 42. Дизайнер формы с панелью свойств формы

Для начала необходимо настроить основные свойства формы так, как требуется для нашего приложения. Ниже перечисляются настраиваемые свойства с кратким описанием функционала. Полное описание свойств формы (и любого другого системного класса) можно найти на сайте <http://msdn.microsoft.com> (MSDN – Microsoft Software Developer Network).

Text – текст, который выводится в заголовке формы. Для большинства визуальных компонентов это свойство присутствует и использу-

ется для статического текста. Следует задать его значение, соответствующее разрабатываемому приложению, например, «Калькулятор».

Icon – значок формы. Значок формы используется как для отображения в левом верхнем углу формы, так и для отображения в панели задач и в окне переключения приложений. Разумно выбрать тот же значок, что и для файла приложения, но это совершенно не обязательно. Если приложение содержит несколько форм, то можно как использовать один значок для всех форм приложения, так и предусмотреть индивидуальные значки для каждой формы. Значок по умолчанию говорит о неспособности разработчика уделить минимальное время дизайну приложения. Следует обратить внимание, что после выбора значка создается дополнительный файл ресурсов формы – `MainForm.resx`, который теперь является частью проекта.

MaximizeBox – управление возможностью окна распахиваться на полный экран. Для калькулятора это явно не требуется, следует выставить значение `False`.

FormBorderStyle – стиль рамки окна. Считаем, что калькулятор не должен изменять размер окна динамически, поэтому следует выбрать один из фиксированных стилей рамки: `FixedSingle`, `Fixed3D` или `FixedDialog`.

Size – размер окна. Устанавливается в дизайнере при помощи изменения размера окна мышью, но можно задать и вручную в численной форме.

StartPosition – стартовое положение окна после запуска программы. Имеется возможность, например, сделать так, чтобы окно после за-

пуска располагалось в центре экрана – для этого следует выбрать значение CenterScreen.

После всех этих изменений окно выглядит следующим образом (Рисунок 43).

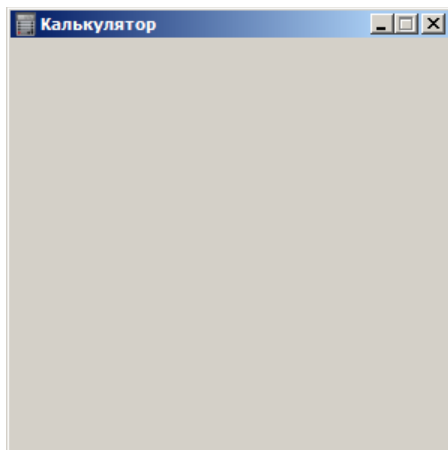


Рисунок 43. Пустая форма калькулятора

2.5. Добавление первых элементов управления

Для добавления элементов управления требуется их выбрать в панели инструментов Toolbox. Если эта панель не отображается автоматически, следует вызвать ее при помощи пункта главного меню «View | Toolbox». Для начала потребуются самые обычные элементы управления, входящие в группу Common Controls. Калькулятор должен иметь индикатор, для чего предлагается использовать текстовое поле TextBox (обычно для статического текста используют метку Label, но TextBox имеет рамку вокруг текста, что больше похоже на обычный калькулятор). Для кнопки имеется соответствующий элемент управления Button.

Добавление нового компонента на форму осуществляется перетаскиванием (drag-n-drop). Для начала следует добавить индикатор и одну кнопку и разместить их на форме красиво (Рисунок 44). При перемещении компонентов следует обратить внимание на графические подсказки, помогающие разместить компоненты с сохранением эргономичных интервалов между ними.

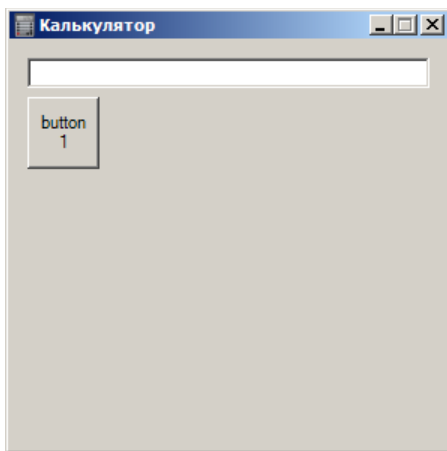


Рисунок 44. Только что добавленные компоненты

Далее следует настроить компоненты. Начнем с текстового поля для индикатора. При его выделении в дизайнере в окне Properties отображаются свойства компонента (Рисунок 45).

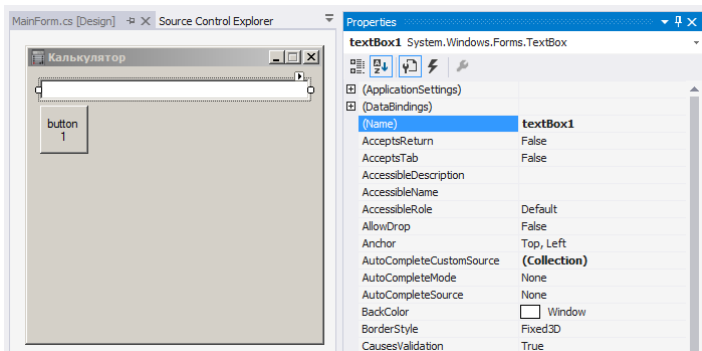


Рисунок 45. Свойства текстового поля

Далее следует задать основные свойства этого компонента. Для начала следует его переименовать – в поле **(Name)** внести имя, соответствующее назначению компонента, например «indicator».

2.5.1. Рекомендации по именованию идентификаторов

Вообще, именование чего бы то ни было в программировании – большая и серьезная тема. В разных языках существуют множество различных подходов к именованию переменных, полей, методов, функций, классов, объектов и всего остального. Среди них можно выделить следующие основные схемы именования:

- **Паскаль** – указание этого стиля оформления идентификатора обозначает, что первая буква заглавная и все последующие первые буквы слов тоже заглавные. Например, **BackColor**, **LastModified**, **DateTime**.
- **Кэмел** – указание этого стиля обозначает, что первая буква строчная, а остальные первые буквы слов заглавные. Например, **borderColor**, **accessTime**, **templateName**.

Ниже приведены общие правила и рекомендации именования идентификаторов, принятые в качестве стандарта при разработке программного обеспечения в компании КРОК:

1. Не использовать префиксы или суффиксы (например, венгерскую нотацию), за исключением случаев, явно указанных в данном документе.
2. Для именования идентификаторов использовать стиль именования Кэмел или Паскаль.
3. По возможности избегать использования сокращений.
4. Не использовать для классов, пространств имен или интерфейсов имена, потенциально или явно конфликтующие со стандартными идентификаторами, а также (по возможности) совпадающие с наименованиями классов и т.д. стандартных библиотек классов Microsoft.
5. Использовать имена, которые ясно и четко описывают предназначение и/или смысл сущности.
6. Запрещается использовать для разных сущностей имена, отличающиеся только регистром букв.
7. Не использовать аббревиатуры в идентификаторах, если только они не являются общепринятыми. Например, `GetWindow`, а не `GetWin`.
8. Широко распространенные акронимы рекомендуется использовать для замены длинных фраз. Например, `UI` вместо `User Interface`.
9. Если имеется идентификатор длиной менее трех букв, являющийся аббревиатурой, то его необходимо записывать заглавными буквами, например `System.IO`, `System.Web.UI`. В остальных случаях идентификатор необходимо записывать в стиле Паскаль или Кэмел, например `Guid`, `Xml`.

Распространенной практикой является включение в имя переменной (свойства, поля) информации об ее типе. Так, для нашего индикатора таким именем стало бы «indicatorTextBox». Но так как современные средства разработки позволяют легко определять тип данных из всплывающей подсказки или контекстного меню, автор считает данную практику избыточной и загромождающей код. Хотя в определенных случаях она, конечно же, имеет право на жизнь.

2.5.2. Настройка текстового поля

Далее необходимо обратить внимание на следующие свойства нашего индикатора:

Font | Size – размер шрифта. Индикатор калькулятора должен быть крупным и заметным, размер шрифта должен быть не менее 16-18 точек (point). После изменения размера шрифта размер индикатора изменится, и потребуются переместить кнопку с тем, чтобы компоненты не перекрывались.

Anchor – задает края формы, к которым привязан размер индикатора. При изменении размера формы (в режиме дизайнера или во время выполнения программы) компонент будет сохранять постоянное расстояние между собой и заданными краями формы. По умолчанию это верхний и левый края формы. Следует добавить к ним правый край формы, в результате значение свойства будет «Top, Left, Right».

Enabled – включение или выключение компонента. Так как непосредственное редактирование текста индикатора не требуется, то надо установить данное свойство в значение False. При этом выделение данного компонента будет невозможно. Следует обратить внимание также на другое свойство, **ReadOnly**, которое служит сходным целям, но несколько иначе. Если текстовое поле помечено

как `ReadOnly`, то его содержимое не может быть изменено (как и в случае `Enabled = False`), но поле может быть выбрано, и в поле будут доступны стандартные операции по копированию его содержимого в буфер обмена Windows. Для наших целей необходимо установить `Enabled = False`, значение `ReadOnly` оставить по умолчанию.

TextAlign – выравнивание текста. Обычно у калькуляторов текст выравнивается по правому краю. Для этого следует установить значение данного свойства, равное «`Right`».

MultiLine – многострочный текст. Так как предполагается, что наш индикатор является однострочным, данное свойство рекомендуется оставить в состоянии по умолчанию, а именно – `False`.

Text – текст, отображаемый в поле. Для начала имеет смысл установить значение «0» (следует понимать, что это строка, а не целое число), хотя это и не является обязательным.

2.5.3. Настройка кнопки

Если посмотреть на стандартный калькулятор Windows, видно, что в верхнем левом углу находится кнопка 7. Поэтому и в данном приложении начнем с кнопки «7». Необходимо настроить, как минимум, следующие свойства кнопки:

(Name) - наименование кнопки (идентификатор). В данном конкретном случае имеет смысл использовать имя вида «`button7`».

Size – размер элемента управления (по аналогии с размером формы). Устанавливается при помощи дизайнера или (в особо ответственных случаях) вручную.

Text – текст, отображаемый на кнопке. Следует установить значение «7» (необходимо обратить внимание на то, что это строка, а не целое число).

Заметим, что до сих пор мы еще не написали ни одной строчки кода.

2.6. Обработка событий элементов управления

Событие – это особая ситуация при выполнении программы, которая автоматически вызывает специальным образом оформленный фрагмент программного кода – обработчик события. Примерами событий являются – нажатие клавиши на клавиатуре, нажатие кнопки мыши, щелчок по кнопке – элементу графического интерфейса, истечение заданного времени, изменение размера окна... В любой программе событий происходит великое множество, но внимания программиста должны привлекать лишь те события, которые требуют нестандартной обработки, то есть написания собственного обработчика. Так, при нажатии на кнопку калькулятора [7] ожидаемым эффектом было бы появление семерки в окошке индикатора. Но по умолчанию нажатие на кнопку никак не обрабатывается. Для этого требуется написать собственный обработчик события нажатия на кнопку.

Для кнопки событие нажатия на кнопку (Click) является базовым событием, обработчик которого создается легко и быстро – следует всего лишь щелкнуть на кнопке в дизайнера два раза. Производится переход к коду обработчика события, а так как обработчик еще не создан, то он создается автоматически в виде следующего кода (Рисунок 46, выделено зеленым):

```
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 WinCalc
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();

            private void button7_Click(object sender, EventArgs e)
            {
            }
        }
    }
}

```

Рисунок 46. Класс формы сразу после создания обработчика события

Так как в данный файл мы еще не попадали, следует пояснить синтаксические конструкции, которые ранее не встречались.

2.6.1. Класс и конструктор класса

Уже достаточно много было сказано про классы, но до сих пор это понятие не было формально введено. Класс – это тип данных, который содержит не только сами данные (поля класса), но и программный код, который эти данные обрабатывает (методы класса). Соединение описания данных и обрабатывающего их кода в рамках одного описания – один из фундаментов объектно-ориентированного подхода. Следует четко различать классы и объекты. Объект – это переменная, тип данных которой является классом. Соответственно, для данного класса могут существовать бесконечное множество объектов. Впрочем, их реальное количество

ограничено тем или иным образом. Физическим образом, если речь идет о доступном объеме оперативной памяти. Логическим образом, если объекты данного класса может существовать в ограниченном (или вовсе единственном) числе.

У каждого класса выделяется особый метод (или методы), который называется конструктор класса. Данный метод вызывается при создании объекта данного класса. Конструктор, не имеющий параметров, называется конструктором по умолчанию. В данном случае конструктор по умолчанию класса `MainForm` содержит единственную строку, в которой производится инициализация (создание и настройка) всех элементов управления данной формы при помощи вызова метода `InitializeComponent`. Заметим, что метод `InitializeComponent` описан во втором файле данного класса и формируется дизайнером формы автоматически.

2.6.2. Описание класса

Как уже было показано выше, ключевое слово `class` начинает описание класса. В данном описании есть две новые конструкции, которые не встречались ранее.

partial – ключевое слово, означающее, что исходный текст данного класса расположен в двух и более файлах. В данном случае такими файлами являются `MainForm.cs` и `MainForm.Designer.cs`. Первый файл – непосредственное творение программиста, а второй файл формируется автоматически в результате редактирования формы в дизайнера. Если рассмотреть данный файл, будет видно, что в процессе интерактивного рисования формы создается достаточно объемный, хотя и тривиальный программный код. Разделение кода формы между двумя файлами позволяет отделить код, написанный вручную, от кода, сформированного автоматически.

Двоеточие в описании класса означает наследование. Конструкция `MainForm : Form` означает, что класс `MainForm` является наследником класса `Form` (точнее, `System.Windows.Forms.Form`). Наследование – один из трех фундаментальных механизмов объектно-ориентированного программирования. Наследовать (быть наследником) – означает иметь все свойства класса-предка, а именно – все поля, все свойства, все методы, все события. Только что созданный класс-наследник ведет себя точно так же, как и его предок. Но после того, как в класс-наследник будут добавлены новые поля, свойства, методы и события, его поведение изменится и будет отличаться от предка в объеме сделанных изменений. Только что созданная главная форма велась точно так же, как и базовый класс `Form`. Но после того, как к ней добавили элементы управления, ее поведение стало уникальным.

Механизм наследования очень мощный, и он будет более подробно раскрыт ниже, когда это будет уместно.

2.6.3. Модификаторы доступа

Современные программы создаются на основе модульного подхода. Соответственно, каждый модуль (будь то проект в целом, класс, поле, метод и др.) должен быть по возможности максимально изолирован от другого модуля. Это называется принципом наименьших привилегий. Доступ к каждому программному объекту должен быть такой, чтобы программа функционировала нормально, но не более того.

public – Один из пяти модификаторов доступа в языке C#. Вначале он применен по отношению к классу `MainForm`. Для классов доступны только два модификатора, изображенные ниже (Рисунок 47). Модификатор `public` означает, что класс доступен везде – в рамках данного проекта (сборки), а также во всех других проектах, которые

захотят использовать данный проект. Альтернативный модификатор – **internal**, который означает, что класс доступен только внутри данного проекта, но не за его пределами. Если модификатор не задан (Рисунок 39), то, в соответствии с принципом наименьших привилегий, используется модификатор **internal**. Следует заметить, что для данного проекта доступ **public** к классу **MainForm** избыточен – использование этой формы за пределами проекта пока не планируется. Поэтому имеет смысл заменить его на **internal**.

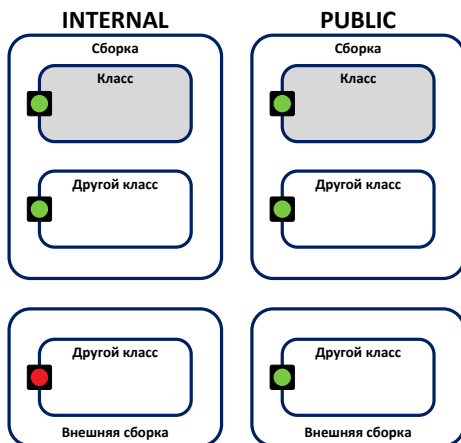


Рисунок 47. Модификаторы доступа к классам и структурам

Модификатор **public**, примененный далее к методу **MainForm** (данный метод называется конструктором, см раздел 2.6.1), описывает его как доступный кому угодно. Если строго следовать принципу наименьших привилегий, его тоже стоит объявить как **internal**.

private – самый строгий модификатор доступа, он применен к обработчику события **button7_Click**. Модификатор ограничивает доступ к этому обработчику – он доступен только внутри того класса, где описан (Рисунок 48). Другие возможные модификаторы доступа для

членов класса также показаны на этом рисунке, они будут рассмотрены, как только в них появится необходимость. Так как доступа более строгого, чем `private`, не существует, менять в данном случае ничего не требуется.

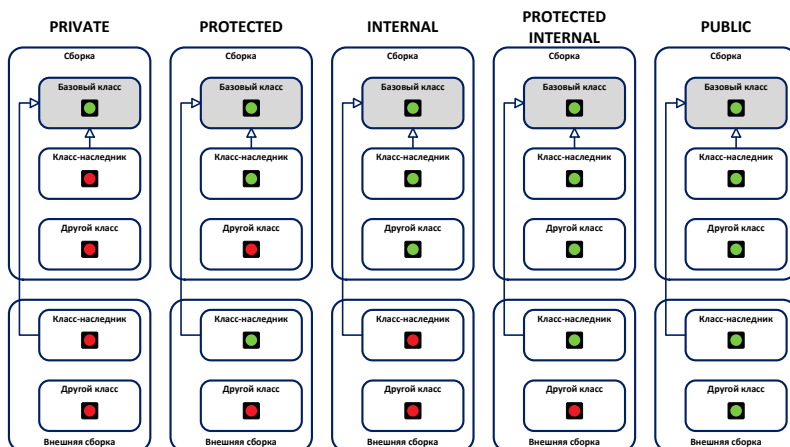


Рисунок 48. Модификаторы доступа к элементам класса (включая вложенные классы)

2.6.4. О пользе комментариев

Выше было приведено много сведений, некоторые из которых уместно было бы включить непосредственно в текст программы в виде комментариев. Комментарии в языке C# имеют три варианта (Рисунок 49):

```
// однострочные комментарии (до конца строки)
/*
Многострочные комментарии - до ближайшей комбинации звездочки и косой черты
*/

/// <summary>
/// XML-комментарии, которые относятся к различным программным артефактам - классам, полям и пр.
```

```
/// </summary>
```

Рисунок 49. Комментарии в языке C#

Если с первыми двумя всё понятно, то третий вариант требует пояснений. При вводе символов «`///`» перед любым описанием структурного элемента языка (пространством имен, классом, полем, методом и пр.) эти три символа автоматически преобразуются в комментарий специального вида (XML documentation-комментарий), который используется для автоматического построения документации, в том числе и для работы механизма IntelliSense.

Ниже приведены требования к комментариям, используемые при разработке программного обеспечения в компании КРОК:

1. Для описания классов и методов используются XML documentation-комментарии.
2. Рекомендуется отделять текст комментария одним пробелом «`//` Текст комментария».
3. Для сложных алгоритмов (методы машинной графики, расчетные алгоритмы и т.д.) должно быть включено его описание или ссылка на источник (внешний документ, наименование стандартного алгоритма и т.д.).
4. Каждый неявный фрагмент кода должен быть прокомментирован.
5. По завершении процесса разработки (при закрытии задачи) из исходного кода должны быть убраны закомментированные блоки кода. Для хранения истории изменений служит TFS. Если по завершении процесса разработки остались закомментированные блоки кода, то необходимо пояснить комментарием, чем это обусловлено.

2.6.5. Код обработчика события

Для первого запуска достаточно поместить в обработчик события следующий код (Рисунок 50, заметим, что код уже снабжен необходимыми комментариями)

```
/// <summary>
/// Обработчик нажатия на кнопку
/// </summary>
/// <param name="sender">Объект, вызвавший событие</param>
/// <param name="e">Параметры события</param>
private void button7_Click(object sender, EventArgs e)
{
    indicator.Text = "7";
}
```

Рисунок 50. Простейший обработчик

Но такой обработчик не обработает даже второе нажатие на кнопку. Поэтому его необходимо изменить для начала следующим образом: `indicator.Text += "7";`

Оператор `+=` прибавляет к содержимому переменной слева результат вычисления выражения справа. Для строк арифметическое сложение заменяется конкатенацией (соединением) строк. Так как `Text` – текстовое поле нашего индикатора, данный код работает ожидаемым образом. Но при этом нигде не хранится число, которое мы собираемся обрабатывать – только его строковое представление. Поэтому самое время подумать о структуре нашего приложения.

Для хранения числа, которое соответствует показаниям индикатора, потребуется переменная. Так как она относится к форме, логично описать ее внутри класса `MainForm`. Соответственно, получаем целочисленное поле (Рисунок 51). Поле описано как `private`, так как доступ к нему требуется только изнутри класса формы.

```
/// <summary>
```

```
/// Значение индикатора  
/// </summary>  
private int x;
```

Рисунок 51. Поле для значения индикатора

Для порядка поле следует инициализировать при помощи оператора $x = 0$. Это можно сделать в двух местах – в конструкторе (тогда оператор имеет полную форму) или сразу при объявлении поля, что дает там объявление вида `private int x = 0`. Следует заметить, что для целочисленных полей 0 – это значение по умолчанию, так что в явной инициализации нет особого практического смысла, но явная инициализация переменных – признак хорошего тона, а в некоторых случаях она требуется обязательно.

Текст обработчика события после этого превращается в следующую конструкцию:

- $x = 10 * x + 7$;
- `indicator.Text = x.ToString();`

Первый оператор достаточно очевиден, а второй требует комментария. Несмотря на то, что x – целое число, это еще и объект типа `System.Int32`. В языке C# абсолютно все классы и структуры, не имеющие явных предков, неявно (то есть без специального описания) наследуют от системного класса `Object`. У класса `Object` описано несколько методов, в данном случае используется метод `ToString`, который преобразует объект в его строковое представление. Естественно, что метод `ToString` класса `Object` не знает, как преобразовать целое число в строку, но перекрытый метод в структуре `System.Int32` таким знанием обладает. Здесь мы впервые сталкиваемся с перекрытием методов, но этот механизм будет раскрыт позже в рамках учебного курса более подробно. Полный текст класса `MainForm` сейчас выглядит следующим образом (Рисунок 52).

```

/// <summary>
/// Главная форма приложения
/// </summary>
internal partial class MainForm : Form
{
    /// <summary>
    /// Значение индикатора
    /// </summary>
    private int x = 0;

    /// <summary>
    /// Конструктор по умолчанию
    /// </summary>
    internal MainForm()
    {
        InitializeComponent();
    }

    /// <summary>
    /// Обработчик нажатия на кнопку
    /// </summary>
    /// <param name="sender">Объект, вызвавший событие</param>
    /// <param name="e">Параметры события</param>
    private void button7_Click(object sender, EventArgs e)
    {
        x = 10 * x + 7;
        indicator.Text = x.ToString();
    }
}

```

Рисунок 52. Класс формы с обработчиком единственной кнопки

2.7. Добавление цифровых элементов управления

В текущем приложении есть всего лишь одна кнопка. Далее имеет смысл добавить остальные цифровые клавиши. Это можно сделать двумя способами:

1. Добавить кнопки непосредственно из панели Toolbox. В этом случае размеры каждой кнопки придется настраивать индивидуально
2. Скопировать существующую кнопку (Ctrl+мышь) и отредактировать ее. В этом случае все свойства (внимание: **все** свойства) кнопки будут скопированы из оригинала. Автор обычно пользуется именно этим методом. Копировать кнопки рекомендуется по одной (это связано с тем, что вновь создаваемая кнопка получает автоматическое имя buttonN, где N – первый незанятый номер).

После создания или копирования кнопки следует отредактировать имя (Name) и свойство Text для нее. Возможный результат после добавления всех цифровых кнопок представлен ниже (Рисунок 53).

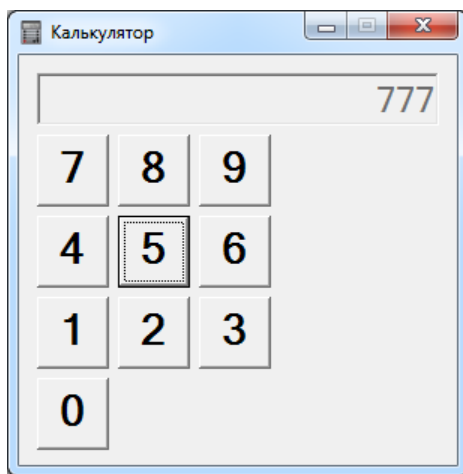


Рисунок 53. Все цифровые кнопки добавлены

2.7.1. Порядок обхода элементов управления (Tab Order)

После запуска программы можно заметить два нюанса. Во-первых, если кнопки добавлялись из панели Toolbox, то нажатие на них не

вызывает никаких действий, но если копировались – выводится цифра «7» независимо от кнопки. Это происходит потому, что при копировании копируются все свойства кнопки, в том числе и ссылка на обработчик события Click. Во-вторых, если попробовать перемещаться между кнопками при помощи клавиши Tab, выяснится, что порядок перемещения не соответствует порядку расположения кнопок на форме (он соответствует порядку их добавления). Обработчики события рассматриваются ниже (раздел 2.8). Для того же, чтобы исправить порядок обхода кнопок (так называемый Tab Order), необходимо сначала вывести соответствующую кнопку на панель инструментов Layout Toolbar (Рисунок 54) – по умолчанию она выключена.

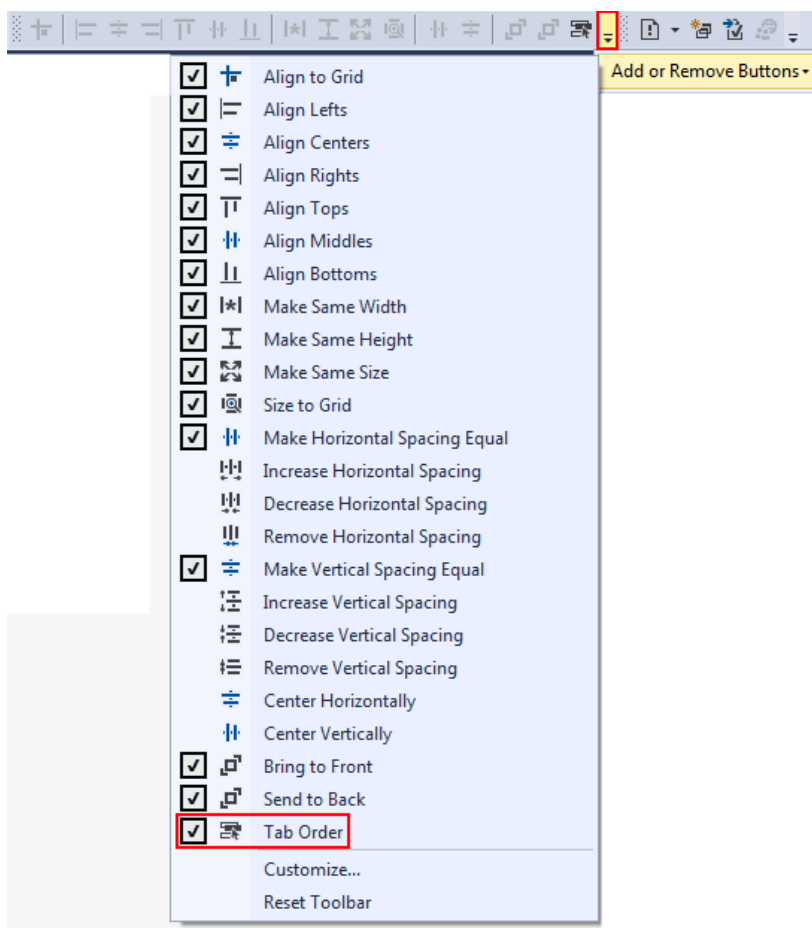


Рисунок 54. Настройка панели Layout Toolbar

Далее следует нажать (активировать) эту кнопку и прощелкать все активные элементы управления (в данном случае только кнопки) в заданном порядке (Рисунок 55). После чего следует снова нажать эту кнопку для выхода из этого режима. Порядок обхода сохраняется в свойстве **TabOrder**, которое есть у каждого компонента формы.

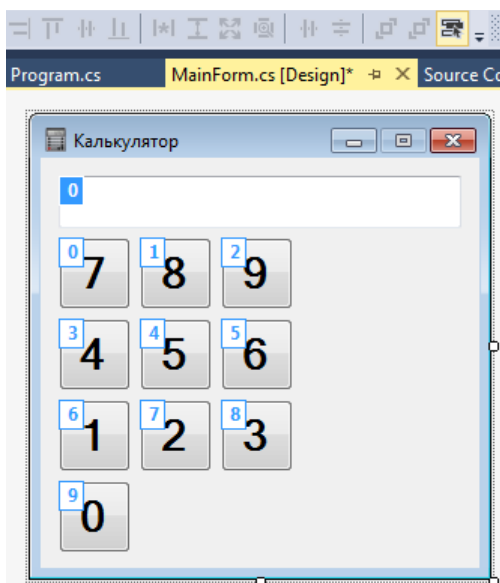


Рисунок 55. Управление порядком обхода компонентов формы

2.8. Универсальный обработчик цифровой кнопки

Самый примитивный способ заставить калькулятор обрабатывать все 10 цифровых клавиш – написать 10 обработчиков. Но если кнопка копировалась, то надо как-то создать индивидуальный обработчик для каждой кнопки. Для этого в дизайне надо в свойствах кнопки выбрать события (Рисунок 56) и перейти к событию Click (следует обратить внимание на обилие различных событий, которые могут случаться с кнопкой).

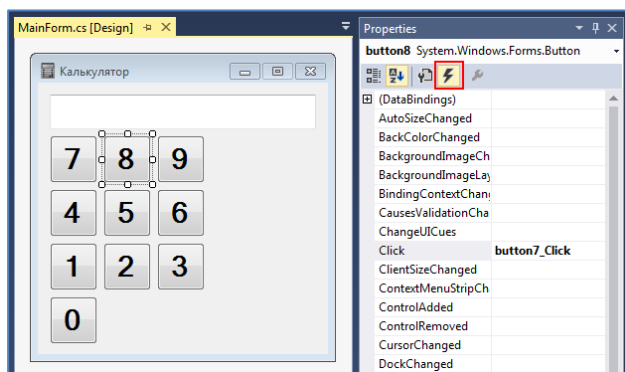


Рисунок 56. Просмотр событий кнопки и их обработчиков

Если отредактировать имя обработчика в этом поле, то автоматически будет создан новый обработчик, который можно было бы реализовать следующим образом (Рисунок 57):

```
private void button8_Click(object sender, EventArgs e)
{
    x = 10 * x + 8;
    indicator.Text = x.ToString();
}
```

Рисунок 57. Обработчик события нажатия на кнопку [8]

Примечание. Если стереть имя обработчика в этом поле, то элемент управления не будет иметь обработчика данного события. Для создания обработчика потребуется или внести его имя вручную, или щелкнуть дважды левой кнопкой мыши на элементе управления в дизайнера. Это знание пригодится впоследствии.

Легко догадаться, что десять почти одинаковых обработчиков не свидетельствуют о высокой квалификации программиста. По факту они отличаются лишь единственным параметром – цифрой, которая написана на кнопке. Но обработчик получает параметры, при по-

мощи которых можно эту цифру определить и написать один обработчик на все возможные цифровые кнопки.

Первым параметром обработчика любого события часто является объект, который данное событие вызвал, как и в данном случае. Проблема в том, что объект `sender` передается в обработчик в виде параметра типа `object` – то есть практически без какой-либо дополнительной информации. Но на самом деле источником события является кнопка, и именно она передается в данный метод в виде параметра `sender`. Впрочем, в этом легко убедиться, используя интерактивный отладчик, встроенный в среду разработки Visual Studio.

Для начала в начале обработчика следует поставить точку останова (клавиша F9 или пункт меню «Debug | Toggle Breakpoint»). После запуска приложения и нажатия кнопки выполнение будет прервано в этой точке останова. В окне Watch (пункт меню «Debug | Windows | Watch | Watch 1») следует ввести выражение `sender.GetType()` и удостовериться в том, что фактический тип параметра – `System.Windows.Forms.Button` (Рисунок 58).

Примечание. Чтобы программа больше не останавливалась на этом месте, следует не забыть очистить точку останова таким же образом, как она была установлена.

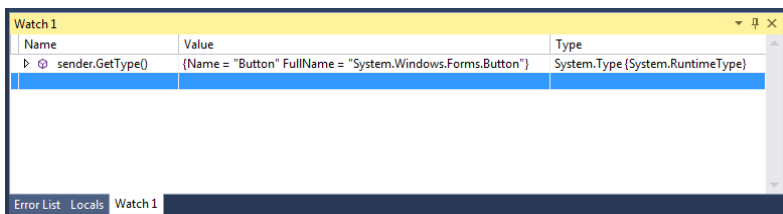


Рисунок 58. Окно слежения за значениями выражений при отладке

Метод `GetType()` – другой полезный метод класса `Object`. Для любой переменной (поля, свойства, параметра) в программе использование этого метода позволяет установить фактический тип значения. Таким образом становится очевидным, что при помощи типа `Object` может быть передано значение любого типа (так как любой тип данных явно или неявно наследует от типа `Object`). Кроме того, становится понятным, что фактический тип данных значения может отличаться от типа данных описания.

Чтобы использовать это новое знание, необходимо осуществить явное приведение типов, то есть каким-то образом объяснить программе, что в переменной `sender` на самом деле скрывается объект класса `Button`. Для этого служит операция явного приведения типов:

- `Button b;`
- `b = (Button)sender;`

Данная последовательность объявляет переменную `b` типа `Button` и инициализирует ее при помощи присваивания с явным приведением типов. Так как переменная `sender` по факту и содержит значение типа `Button`, приведение типов осуществляется без ошибки. Следует понимать, что в данном случае не осуществляется преобразование типа данных (мы не делаем из абстрактного объекта кнопку), в отличие, например, от метода `ToString`, который осуществляет именно преобразование значения одного типа в другой тип данных. Более кратко эта последовательность может быть записана следующим образом:

- `Button b = (Button)sender;`

Далее следует определить, какая именно кнопка была нажата. В данном случае существуют три возможных способа:

1. Проанализировать текст, написанный на кнопке
2. Проанализировать название (идентификатор) кнопки
3. Использовать тег кнопки (свойство Tag)

Казалось бы, использовать текст, написанный на кнопке, проще всего. Но имеет смысл подумать – а всегда ли у нас на кнопке будет написана цифра? Если будет реализовываться калькулятор для программистов, то на кнопке может появиться шестнадцатеричная цифра (то есть буква). Кроме того, на кнопке может и не быть текста – вместо текста можно использовать графический значок. Становится понятно, что текст в качестве идентификатора – плохая идея.

Название кнопки кажется более разумным выбором – в конце концов, название определяется самим программистом. Можно взять название, отрезать последний символ и использовать в качестве номера кнопки. В принципе для некоторых случаев такой подход оправдан, но есть способ лучше.

У каждого элемента управления формы есть специальное свойство – Tag (тег) – для хранения пользовательской информации. Разумно для хранения цифрового значения кнопки использовать именно его. Заполняем свойство Tag для всех десяти кнопок (Рисунок 59).

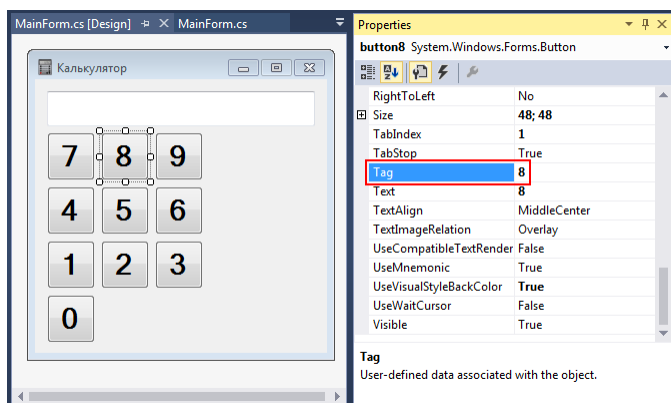


Рисунок 59. Заполнение свойства Tag

Далее необходимо получить значение тега. Так как дизайнер формы вводит значение тега в строковом виде, то и в коде он возвращается как строка (хотя имеет тип object). Снова используется явное приведение типа:

- `string tag = (string)b.Tag;`

От значения тега следует перейти к цифре. Конструкция вида `int digit = (int)tag` не сработает, так как тут уже требуется не приведение типа, а его преобразование – из строки надо сделать число. Используется системный метод `Parse`:

- `int digit = int.Parse(tag);`

Наконец, используем полученную переменную в выражении оператора присваивания.

Так как обработчик теперь будет обрабатывать не только нажатие кнопки [7], но и всех остальных кнопок, имеет смысл его переименовать. Для этого следует использовать пункт контекстного меню `Rename` (или клавишу `F2`) на имени обработчика – только в этом

случае все ссылки на обработчик будут переименованы автоматически. Если просто исправить имя, то придется исправлять его везде, где на него есть ссылки в автоматически создаваемом коде формы.

После всех модификаций обработчик выглядит следующим образом (Рисунок 60):

```
/// <summary>
/// Обработчик нажатия на кнопку
/// </summary>
/// <param name="sender">Объект, вызвавший событие</param>
/// <param name="e">Параметры события</param>
private void buttonDigit_Click(object sender, EventArgs e)
{
    // Кнопка, нажатая пользователем
    Button b = (Button)sender;
    // Тег кнопки в виде строки
    string tag = (string)b.Tag;
    // Цифра в виде целого числа
    int digit = int.Parse(tag);
    // Изменение значения
    x = 10 * x + digit;
    // Обновление индикатора
    indicator.Text = x.ToString();
}
```

Рисунок 60. Обработчик с использованием тега кнопки

2.9. Первое действие: сложение двух чисел

Чтобы реализовать хотя бы одно арифметическое действие, нужно проделать достаточно много операций. Для начала требуется добавить еще две кнопки к главной форме – кнопку «+» и кнопку «=». Для определенности предлагается назвать эти кнопки `buttonAdd` и `buttonResult` соответственно. В результате дизайн формы может быть, например следующим (Рисунок 61):

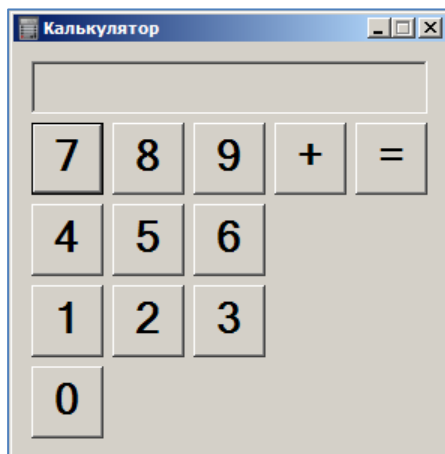


Рисунок 61. Форма с первым арифметическим действием

После добавления новых элементов управления (как в этом, так и в последующих случаях) рекомендуется заново определить порядок обхода элементов управления (см. раздел 2.7.1).

Далее следует создать два новых обработчика событий для обеих кнопок.

2.9.1. Обработчик операции сложения

Для того чтобы реализовать операцию сложения, необходимо выполнить две задачи:

1. где-то запомнить первый аргумент операции;
2. дать возможность пользователю перейти к вводу второго аргумента.

Для реализации первой задачи потребуется еще одно поле класса, назовем его y :

- `private int y;`

Для того, чтобы зафиксировать ситуацию, при которой необходимо начать ввод нового числа, а не добавлять цифры к уже существующему, требуется ввести некий логический признак. Назовем его `newNumber`:

- `private bool newNumber = true;`

В данном объявлении использован новый тип данных – `bool`. Это логический тип данных, который может принимать всего два значения – `true` (истина) и `false` (ложь). Так как сразу после запуска калькулятора он должен перейти в режим ввода нового числа, поле инициализируется значением `true`.

Код обработчика клавиши «+» при этом выглядит следующим образом (Рисунок 62):

```
private void buttonAdd_Click(object sender, EventArgs e)
{
    y = x;
    newNumber = true;
}
```

Рисунок 62. Код обработчика операции сложения

2.9.2. Модификация обработчика цифровых клавиш

Теперь необходимо модифицировать текст обработчика цифровых клавиш. Требуется проверить значение признака `newNumber` и изменять значение поля `x` в зависимости от этого признака различным образом. При этом важно не забыть сбросить признак в значение `false` после анализа. Модифицированный фрагмент обработчика приведен ниже (Рисунок 63). Этот фрагмент должен быть вставлен вместо фрагмента, выделенного серым фоном (Рисунок 60).

```
if (newNumber)
{
```

```

        // Начало ввода нового значения
        x = digit;
        // Сброс признака ввода нового числа
        newNumber = false;
    }
    else
    {
        // Изменение значения
        x = 10 * x + digit;
    }
}

```

Рисунок 63. Код обработчика цифровых клавиш

Данный фрагмент кода содержит использование условного оператора if. Синтаксис оператора аналогичен многим другим языкам программирования (например, C) и состоит из ключевого слова if, логического условия, заключенного в скобки, блока кода, который выполняется при истинности условия, и необязательного блока else, который выполняется, если условие ложно. Следует обратить внимание на разницу в операторах присваивания поля x.

2.9.3. Обработчик кнопки вычисления

Наконец, надо реализовать вычисление суммы двух чисел. Фрагмент кода, обеспечивающий это непростое действие, приведен ниже (Рисунок 64). Следует обратить внимание на два основных момента

1. После присваивания значения полю x требуется обязательное обновление свойства Text объекта indicator;
2. После вычисления калькулятор переходит в режим ввода нового числа.

```

// Вычисление суммы двух чисел
x = x + y;
// Обновление индикатора
indicator.Text = x.ToString();
// Переход к вводу нового числа
newNumber = true;

```

2.9.4. Поля и свойства

Теперь имеет смысл обратить внимание на тот факт, что каждый раз после изменения значения поля `x` осуществляется изменение значения свойства `indicator.Text`. Причем, похоже, это является правилом. Одним из фундаментальных принципов программирования является устранение дублирования кода путем использования каких-либо программных блоков – подпрограмм, функций, процедур и пр. В терминах языка C# в данном конкретном случае следует осуществить переход от использования поля к использованию свойства.

Свойство – это элемент данных, доступ к которому осуществляется при помощи программного кода. В этом его фундаментальное отличие от поля или переменной. Поле или переменной – это именованная ячейка памяти. Свойство – это программный код, под которым может лежать, а может и не лежать переменная для хранения данных. Свойство для программиста ведет себя как переменная, то есть его можно использовать и в левой, и в правой части оператора присваивания. Но реализация свойства может иметь дополнительные эффекты, чем сейчас мы и воспользуемся.

Для начала поле `x` переименуется в `_x` (в языке программирования C# символ подчеркивания используется на правах буквы). Переименование следует осуществить вручную, простым редактированием текста, а не через пункт контекстного меню `Rename`, как делалось ранее. Тому есть причина – далее будет создано свойство с тем же самым именем – `x`. При этом поле уже не инициализируется нулем – это будет сделано с использованием свойства:

- `private int _x;`

Свойство содержит два основных элемента – get и set. При помощи get осуществляется чтение значения свойства, и такой блок кода должен всегда завершаться оператором return, возвращающим значение свойства. Блок get вызывается при использовании свойства в правой части оператора присваивания и в аналогичных случаях (передача в качестве параметра и пр.).

Блок set используется для сохранения значения свойства и имеет неявный параметр – значение свойства value. Блок set вызывается при использовании свойства в левой части оператора присваивания. Побочным эффектом свойства является обновление индикатора. Теперь можно исключить обновление индикатора везде, где присваивается значение x.

```
private int x
{
    get
    {
        return _x;
    }
    set
    {
        _x = value;
        // Обновление индикатора
        indicator.Text = _x.ToString();
    }
}
```

Рисунок 65. Код свойства

Так как мы отказались от инициализации поля, следует перейти к инициализации свойства. Для этого в конструктор формы добавляется простой оператор присваивания:

- `x = 0;`

Следует обратить внимание на то, что после этого в поведении калькулятора будет наблюдаться небольшое изменение. Раньше при запуске программы индикатор был пустым (содержал пустую строку). Теперь, так как мы инициализируем не поле, а свойство, сразу после запуска индикатор будет отображать ноль.

Итоговый код формы после всех этих модификаций, снабженный необходимыми комментариями, приведен ниже (Рисунок 66).

```
internal partial class MainForm : Form
{
    /// <summary>
    /// Поле для хранения значения индикатора
    /// </summary>
    private int _x;

    /// <summary>
    /// Значение индикатора
    /// </summary>
    private int x
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
            // Обновление индикатора
            indicator.Text = _x.ToString();
        }
    }

    /// <summary>
    /// Второй аргумент
    /// </summary>
    private int y;

    /// <summary>
    /// Признак ввода нового числа
    /// </summary>
    private bool newNumber = true;
```



```

/// <summary>
/// Конструктор по умолчанию
/// </summary>
internal MainForm()
{
    InitializeComponent();
    // Начальное значение индикатора
    x = 0;
}

/// <summary>
/// Обработчик нажатия на цифровую кнопку
/// </summary>
/// <param name="sender">Объект, вызвавший событие</param>
/// <param name="e">Параметры события</param>
private void buttonDigit_Click(object sender, EventArgs e)
{
    // Кнопка, нажатая пользователем
    Button b = (Button)sender;
    // Тег кнопки в виде строки
    string tag = (string)b.Tag;
    // Цифра в виде целого числа
    int digit = int.Parse(tag);
    // Проверка на начало ввода нового числа
    if (newNumber)
    {
        // Начало ввода нового значения
        x = digit;
        // Сброс признака ввода нового числа
        newNumber = false;
    }
    else
    {
        // Изменение значения
        x = 10 * x + digit;
    }
}

/// <summary>
/// Обработка нажатия на кнопку арифметического действия
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonAdd_Click(object sender, EventArgs e)

```

```

{
    // Запомним второй аргумент операции
    y = x;
    // Переход к вводу нового числа
    newNumber = true;
}

/// <summary>
/// Обработка нажатия на кнопку вычисления
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonResult_Click(object sender, EventArgs e)
{
    // Вычисление суммы двух чисел
    x = x + y;
    // Переход к вводу нового числа
    newNumber = true;
}
}

```

Рисунок 66. Оптимизированный код формы

2.10. Остальные арифметические действия

Для реализации остальных бинарных операций - вычитания, умножения и деления - следует добавить еще три кнопки (Рисунок 67).

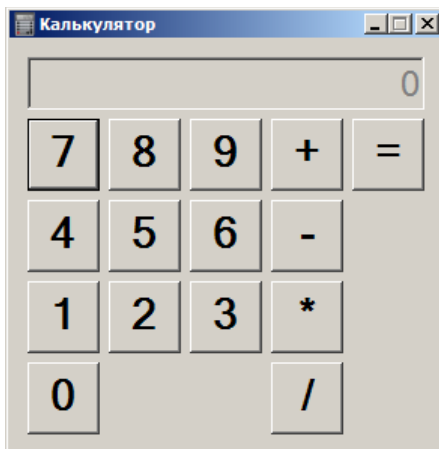


Рисунок 67. Четыре арифметические кнопки

При этом возникает новая задача – необходимо запомнить нажатую операцию, так как само вычисление будет осуществляться при нажатии кнопки «=». Для этого можно было бы завести еще одно целочисленное поле, в котором сохранить код операции (например, 1 для сложения, 2 для вычитания и так далее), но предлагается использовать более красивый, удобный и наглядный способ – перечислимые типы данных.

2.10.1. Перечислимые типы данных

Перечислимый тип данных – это фактически набор именованных целочисленных констант. Но значение констант не имеет существенного значения. Основной смысл перечисления – некоторый фиксированный набор вариантов, каждый из которых имеет собственное название. В нашем случае у нас есть четыре действия арифметики – сложение, вычитание, умножение и деление. Для того, чтобы запомнить одно из четырех действий, предлагается использовать перечислимый тип, который разумно назвать *Operation*.

Для размещения перечислимого типа предлагается использовать отдельный одноименный файл, который следует добавить к проекту при помощи пункта контекстного меню, которое открывается по правой клавише мыши на файле проекта. Для перечислимых типов нет отдельного шаблона, поэтому следует выбрать пункт «Add | Class». В появившемся диалоговом окне необходимо ввести название перечислимого типа – Operation (Рисунок 68).

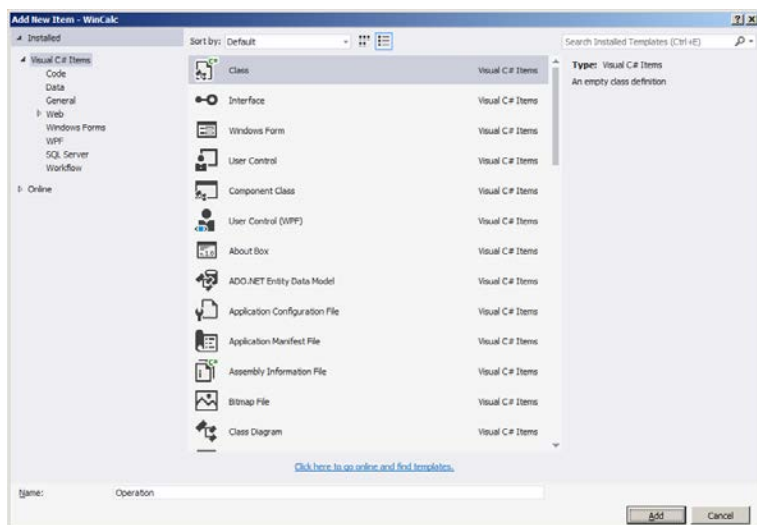


Рисунок 68. Добавление нового класса или перечислимого типа

Вместо описания класса требуется описать перечислимый тип при помощи ключевого слова `enum` (Рисунок 69). Следует обратить внимание, что помимо четырех действий арифметики в перечисление включено еще одно значение – `None` – которое будет использоваться для случаев, когда действие еще не задано. Кроме того, имеет смысл добавить необходимые комментарии как к типу в целом, так и к каждому конкретному элементу.

```
internal enum Operation
```

```
{  
    None,  
    Addition,  
    Subtraction,  
    Multiplication,  
    Division  
}
```

Рисунок 69. Описание перечислимого типа

Альтернативой использования перечислимого типа в данном случае могло бы стать использования набора именованных констант, описанных в рамках статического класса (Рисунок 70). Фундаментальным отличием этого варианта от предыдущего является то, что для хранения целочисленной константы требуется поле или переменная целочисленного типа (int), и данная переменная может содержать значения не только от 0 до 4, но и любые другие числа. А в случае использования перечислимого типа язык гарантирует, что кроме явно описанных, никаких других посторонних значений поле или переменная типа Operation содержать в принципе не может. Поэтому в рамках данного проекта будет использоваться именно перечислимый тип.

```
internal static class OperationClass  
{  
    internal const int None = 0;  
    internal const int Addition = 1;  
    internal const int Subtraction = 2;  
    internal const int Multiplication = 3;  
    internal const int Division = 4;  
}
```

Рисунок 70. Набор целочисленных констант

Второй особенностью перечислимого типа является то, что он не нуждается в явной нумерации значений. Несмотря на то, что с каждым перечислением связано его целочисленное значение (нумерация начинается с нуля), зачастую знать конкретные значения этих

целых чисел не требуется, как и в данном случае. Тем не менее, при необходимости можно задать и конкретное целочисленное значение, и оно может не совпадать с значением, которое было бы присвоено по умолчанию, например (Рисунок 71):

```
internal enum Operation
{
    None,
    Addition = 2,
    ...
}
```

Рисунок 71. Явное задание значения перечисления

2.10.2. Преобразование из строки в перечислимый тип

Для хранения операции внутри класса формы описывается очередное поле с инициализацией значения:

- `private Operation operation = Operation.None;`

Как и ранее, нет смысла описывать для арифметических кнопок отдельные обработчики. Вместо этого предлагается применить тот же метод, что и с цифровыми кнопками – заполнение тега у каждой кнопки и определение кнопки, исходя из параметров вызова события.

С заполнением значения тега все должно быть просто. В качестве значений тега предлагается использовать те же идентификаторы, что и в перечислимом типе (Рисунок 72).

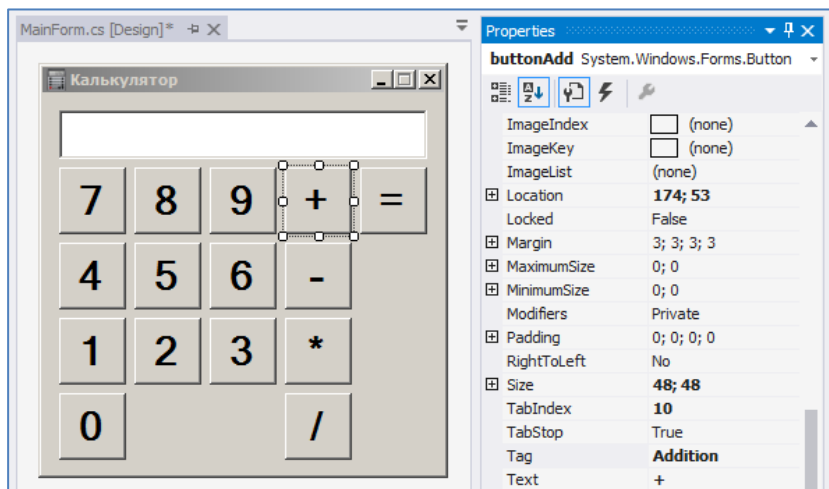


Рисунок 72. Значение тега арифметической кнопки

Обработчик тоже имеет смысл переименовать в соответствии с назначением - `buttonOperation_Click`. И в его код конечно же придется внести изменения. Начало будет аналогичным обработчику цифровой кнопки – требуется получить строковое значение тега:

- `Button b = (Button)sender;`
- `string tag = (string)b.Tag;`

Далее требуется осуществить преобразование строки в значение перечислимого типа. Это можно осуществить двумя способами.

Первый способ – использование метода `Parse`, который описан в системном типе `System.Enum`, обеспечивающем поддержку всех перечислимых типов. Метод возвращает значение типа `object`, так что потребуется явное приведение типов. Метод принимает два аргумента – описание перечислимого типа и строковое значение для преобразования. Описание любого типа данных имеет свой системный тип данных `System.Type`. Выше мы уже сталкивались с этим ти-

пом, когда определяли фактический тип данных параметра sender (Рисунок 58). В данном случае требуется получить объект типа System.Type не по переменной, а по самому типу данных. Для этого используется специальное ключевое слово typeof. В результате получается конструкция следующего вида:

- `operation = (Operation)Enum.Parse(typeof(Operation), tag);`

Второй способ – использования обобщенного метода TryParse. Любой обобщенный метод содержит код, не зависящий от конкретного типа данных (фактически тип данных передается как неявный параметр). Данный метод возвращает логическое значение – результат выполнения преобразования (успешно или нет). Результат самого преобразования возвращается через второй аргумент, который объявлен как возвращаемый при помощи специального ключевого слова out. В языке C# требуется явное указание случаев, когда метод может изменять значение своего параметра. Это делает код более наглядным и избавляет от ситуаций, когда метод изменяет значение параметра, а программист об этом не подозревает. Этот вариант выглядит следующим образом:

- `Enum.TryParse<Operation>(tag, out operation);`

Данный вариант кода может быть упрощен в связи с тем, что тип данных, который является параметром обобщенного метода, задан в явной форме своим вторым аргументом. Так что код может быть еще немного упрощен:

- `Enum.TryParse(tag, out operation);`

2.10.3. Оператор множественного выбора

Теперь следует изменить обработчик кнопки «=» с тем, чтобы в зависимости от значения поля operation выполнялись бы различные

действия. Напрашивается многократное использование условного оператора (Рисунок 73).

```
if (operation == Operation.Addition)
{
    x = y + x;
}
else if (operation == Operation.Subtraction)
{
    x = y - x;
}
else if (operation == Operation.Multiplication)
{
    x = y * x;
}
else if (operation == Operation.Division)
{
    x = y / x;
}
```

Рисунок 73. Многократное ветвление

Язык предлагает альтернативную синтаксическую конструкцию – оператор множественного выбора switch (Рисунок 74). Оператор осуществляет переход непосредственно на метку, соответствующую значению переменной селектора. Выполняемый код обязан завершиться оператором break, который осуществляет выход из блока. Отсутствие оператора break является не логической ошибкой (как в языке программирования C), а синтаксической ошибкой – такая программа просто не будет скомпилирована.

```
switch (operation)
{
    case Operation.Addition:
        x = y + x;
        break;
    case Operation.Subtraction:
        x = y - x;
        break;
    case Operation.Multiplication:
```

```

        x = y * x;
        break;
    case Operation.Division:
        x = y / x;
        break;
}

```

Рисунок 74. Использование оператора switch

Следует обратить внимание, что операции, зависящие от порядка операндов (вычитание и деление) описаны так: $x = y - x$, а не так: $x = x - y$. Если внимательно разобрать порядок выполнения программы, то очевидно, что первый аргумент (вычитаемое или делимое) к моменту вычисления результата будет размещаться в поле y .

В результате код формы к текущему моменту выглядит следующим образом (Рисунок 75).

```

internal partial class MainForm : Form
{
    /// <summary>
    /// Поле для хранения значения индикатора
    /// </summary>
    private int _x;

    /// <summary>
    /// Значение индикатора
    /// </summary>
    private int x
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
            // Обновление индикатора
            indicator.Text = _x.ToString();
        }
    }
}

```

```

/// <summary>
/// Второй аргумент
/// </summary>
private int y;

/// <summary>
/// Признак ввода нового числа
/// </summary>
private bool newNumber = true;

/// <summary>
/// Операция
/// </summary>
private Operation operation = Operation.None;

/// <summary>
/// Конструктор по умолчанию
/// </summary>
internal MainForm()
{
    InitializeComponent();
    // Начальное значение индикатора
    x = 0;
}

/// <summary>
/// Обработчик нажатия на цифровую кнопку
/// </summary>
/// <param name="sender">Объект, вызвавший событие</param>
/// <param name="e">Параметры события</param>
private void buttonDigit_Click(object sender, EventArgs e)
{
    // Кнопка, нажатая пользователем
    Button b = (Button)sender;
    // Тег кнопки в виде строки
    string tag = (string)b.Tag;
    // Цифра в виде целого числа
    int digit = int.Parse(tag);
    // Проверка на начало ввода нового числа
    if (newNumber)
    {
        // Начало ввода нового значения
        x = digit;
        // Сброс признака ввода нового числа
        newNumber = false;
    }
}

```

```

    }
    else
    {
        // Изменение значения
        x = 10 * x + digit;
    }
}

/// <summary>
/// Обработка нажатия на кнопку арифметического действия
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonOperation_Click(object sender, EventArgs e)
{
    // Кнопка, нажатая пользователем
    Button b = (Button)sender;
    // Тег кнопки в виде строки
    string tag = (string)b.Tag;
    // Запомним операцию
    Enum.TryParse(tag, out operation);
    // Запомним первый аргумент операции
    y = x;
    // Переход к вводу нового числа
    newNumber = true;
}

/// <summary>
/// Обработка нажатия на кнопку вычисления
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonResult_Click(object sender, EventArgs e)
{
    switch (operation)
    {
        case Operation.Addition:
            // Вычисление суммы двух чисел
            x = y + x;
            break;
        case Operation.Subtraction:
            // Вычисление разности двух чисел
            x = y - x;
            break;
        case Operation.Multiplication:

```

```

        // Вычисление разности двух чисел
        x = y * x;
        break;
    case Operation.Division:
        // Вычисление разности двух чисел
        x = y / x;
        break;
    }
    // Переход к вводу нового числа
    newNumber = true;
}
}

```

Рисунок 75. Реализация четырех арифметических действий

2.11. Добавление главного меню формы

Пришло время снабдить наше приложение главным меню. Для этого снова понадобится панель Toolbox, в этот раз необходимые компоненты найдутся в разделе Menus and Toolbars. Меню реализуется при помощи компонента MenuStrip. После добавления меню необходимо сразу же создать как минимум один раздел меню путем заполнения названия пункта. Первый раздел традиционно называется «Файл», а единственным пунктом в этом разделе станет пункт меню «Выход». После добавления меню придется аккуратно передвинуть уже созданные компоненты ниже, так как автоматически этого не произойдет. Кроме того, добавленный компонент меню будет отображен в нижней части дизайнера, так что его вид несколько изменится (Рисунок 76).

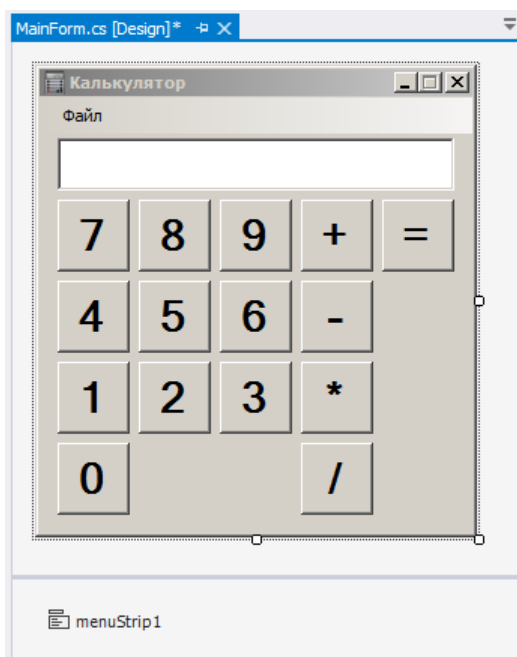


Рисунок 76. Форма после добавления меню

Вновь созданный компонент имеет имя `menuStrip1`. Для начала его следует переименовать. Так как другие меню пока не планируются, предлагается просто назвать его «`menu`». Разделы и пункты меню – это отдельные объекты, у которых тоже есть собственные имена. Автоматическое имя формируется исходя из текста, введенного при создании меню, так что имя пункта меню «Файл» по умолчанию будет `toolStripMenuItem1`. Рекомендуется переименовать все пункты меню, используя только английские буквы для создания идентификаторов.

Далее имеет смысл отредактировать название пунктов меню (свойство `Text`), используя знак «&» для обозначения символа, при помощи которого пункт меню можно выбрать при помощи нажатия на

буквенную комбинацию Alt+буква. Для раздела «Файл» таким же образом получается значение свойства Text, равное «&Text».

Далее по аналогии следует добавить еще один раздел меню – «Правка», содержащий два пункта – «Копировать» и «Вставить». Для этих двух пунктов кроме редактирования свойства Text рекомендуется также заполнить дополнительное свойство ShortcutKeys – клавиатурную комбинацию для вызова данной операции. Традиционно для операции «Копировать» это Ctrl+C, а для операции «Вставить» - Ctrl+V (Рисунок 77).

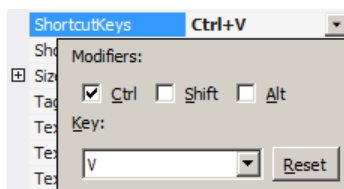


Рисунок 77. Назначение клавиатурной комбинации

Кроме того, для этих пунктов меню имеет смысл задать и собственные значки. Использование типовых значков для типовых функций повышает дружелюбность интерфейса за счет узнавания. В то же время использование нестандартных значков для типовых функций приводит к противоположному результату. Для того чтобы снабдить пункт меню значком, требуется заполнить его свойство Image. Следует обратить, что в данном случае используется не значок в формате ICO, а растровое графическое изображение в одном из следующих форматов: BMP, JPEG, PNG, GIF. Возможно также применение векторного графического изображения в формате WMF.

Так как в приложении появляется несколько изображений, разумно завести отдельный каталог для хранения всей графики. Это можно сделать при помощи пункта контекстного меню «Add | New Folder»

на файле проекта. Далее в этот каталог имеет смысл добавить необходимые графические файлы. После чего уже заполнять свойство Image. При выборе этого свойства отображается диалоговое окно (Рисунок 78).

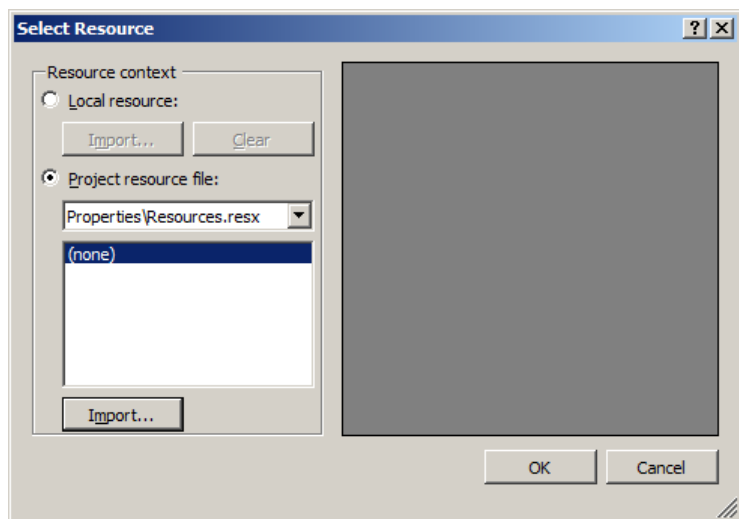


Рисунок 78. Выбор графического изображения (ресурса)

Так как пока никаких изображений в ресурсах проекта не зарегистрировано, следует добавить требуемый файл при помощи кнопки Import и выбрать его при помощи кнопки OK. В результате оформление приложения станет несколько симпатичнее (Рисунок 79).

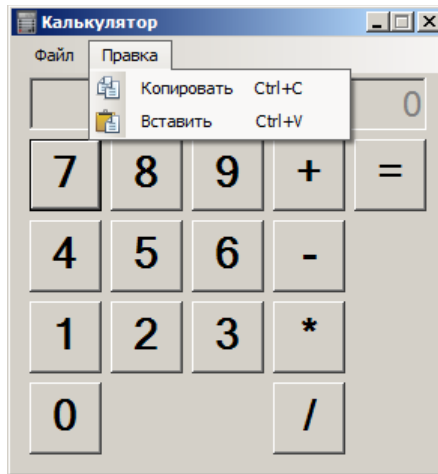


Рисунок 79. Меню со значками

2.12. Обработка событий меню формы

Для того чтобы созданное меню заработало, необходимо создать обработчики пунктов меню. Для начала следует создать обработчик пункта меню «Выход». Это осуществляется уже описанным выше образом – двойным щелчком по пункту меню. В тексте обработчика следует поместить вызов метода закрытия формы, что приведет и к завершению работы приложения, так как эта форма – главная и единственная:

```
Close();
```

Далее следует создать обработчики для двух пунктов раздела «Правка». Для их реализации необходимо ознакомиться с базовыми операциями с буфером обмена Windows.

2.12.1. Буфер обмена Windows

Для работы с буфером обмена предназначен специальный класс `System.Windows.Forms.Clipboard`. Так как буфер обмена существует в

единственном экземпляре, то все методы у данного класса статические, то есть отдельное создание экземпляра класса (объекта) не требуется.

Для того чтобы скопировать содержимое индикатора в буфер обмена Windows в текстовом формате, достаточно этой простой конструкции (Рисунок 80):

```
private void copyToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Clipboard.SetText(indicator.Text);
}
```

Рисунок 80. Реализация обработчика события копирования в буфер обмена

Реализация метода вставки более сложная. Прежде всего, необходимо проверить, что буфер обмена действительно содержит текст с помощью вызова метода `Clipboard.ContainsText()`, который возвращает логическое значение. Далее следует выполнить попытку преобразования строки в число (подобно тому, как делалось ранее при обработке поля Tag) при помощи метода `int.TryParse`. В данном случае приходится использовать дополнительную переменную, так как свойство (в отличие от поля) не может быть передано с модификатором `out` (Рисунок 81). Следует обратить внимание, что идентификатор `value` используется как имя переменной. Вне описания свойства данный идентификатор не является зарезервированным и может быть использован как имя обычной переменной.

```
private void pasteToolStripMenuItem_Click(object sender,
EventArgs e)
{
    if (Clipboard.ContainsText())
    {
        int value;
        if (int.TryParse(Clipboard.GetText(), out value))
        {
            x = value;
        }
    }
}
```

```

    }
}
}

```

Рисунок 81. Реализация обработчика события копирования в буфер обмена

2.12.2. Исключение и обработка ошибок

Вроде бы код, описанный в предыдущем разделе, вполне верный и работоспособный. Но при попытке выполнить его из среды разработки выполнение блокируется и выводится сообщение о необработанном исключении (Рисунок 82). Если же запустить приложение самостоятельно, при помощи исполняемого файла, результаты будут несколько другими, но сходными (Рисунок 83).

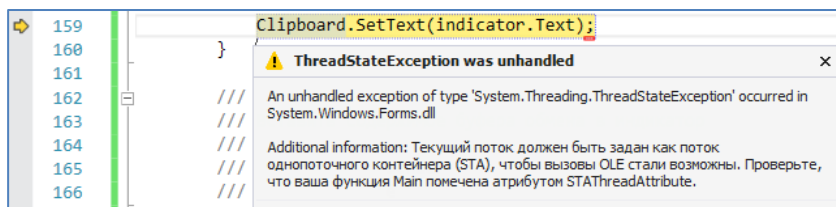


Рисунок 82. Перехват исключения в среде разработки

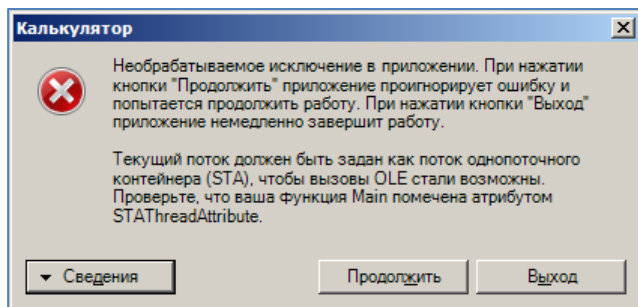


Рисунок 83. Необрабатываемое исключение в приложении

В нашем приложении возникло первое исключение (exception). Исключение – это способ, при помощи которого программный код на языке C# может сообщить о нештатной ситуации, например, о деле-

нии на ноль (кстати, делить на ноль мы тоже еще не пробовали). Если во время выполнения кода возникает нестандартная ситуация, есть возможность создания исключения. В С# используется термин «выброс» (throw), то есть исключение выбрасывается. Далее происходит поиск обработчиков исключения, которые могут поймать исключение данного типа. Поиск происходит сначала в том фрагменте кода, где было выброшено исключение, потом в коде, который вызвал данный код и так далее – этот процесс называется развинчиванием стека (stack unwinding). Пока все это выглядит достаточно загадочно, но далее будет создан обработчик исключения, и дело не-сколько прояснится.

Для обработки исключений используется конструкция try...catch. В самом простейшем случае обработка может выглядеть следующим образом (Рисунок 84):

```
try
{
    Clipboard.SetText(indicator.Text);
}
catch
{
}
```

Рисунок 84. Простейший обработчик исключения

Если в процессе выполнения кода, расположенного внутри блока try, возникает любое исключение, управление передается в блок catch. Так как он пустой, то исключение обрабатывается без всяких видимых эффектов – то есть никакого сообщения об ошибке не возникает, операция просто не выполняется. Конечно же, такое поведение для любого уважающего себя приложения совершенно недопустимо. Как минимум, сообщение об ошибке следует вывести на экран. Для этого используется специальный класс MessageBox и его статический метод Show (Рисунок 85). Метод имеет очень много ва-

риантов, в данном случае используется достаточно распространенный вызов с четырьмя параметрами – текстом сообщения, текстом заголовка окна, кнопкой ОК и значком предупреждения.

```
try
{
    Clipboard.SetText(indicator.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message, "Калькулятор",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
}
```

Рисунок 85. Типовой обработчик исключения

Следует обратить внимание, что конструкция `catch` несколько изменилась – в ней появился параметр. Это как раз и есть исключение, которое тут ловится. Тип `System.Exception` – это корневой класс, от которого наследуются все исключения языка C#. Таким образом, данная конструкция ловит абсолютно все исключения, которые возникают в процессе выполнения блока кода `try`. Пойманное исключение помещается в параметр `ex` (обычно используется именно это имя, но это совершенно не обязательно), и к этому параметру можно обратиться для того, чтобы определить дополнительные сведения о создавшейся нештатной ситуации. Текстовое сообщение об исключении содержится в свойстве `Message`, и наш обработчик выводит его на экран (Рисунок 86).

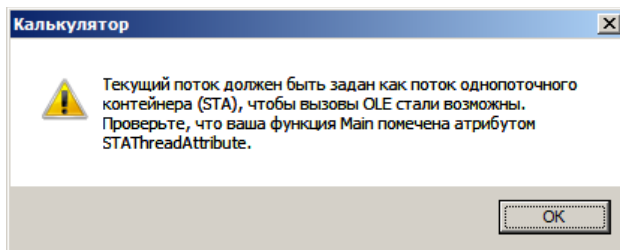


Рисунок 86. Результат вызова метода `MessageBox.Show`

Обработчик исключений, аналогичный описанному выше (Рисунок 85), надо вставить во все обработчики свойств. Необходимость этого обусловлена тем, что обработчики вызываются из кода, внешнего по отношению к приложению, и если исключение не будет обработано нашим собственным кодом, то оно попадет в инфраструктуру Microsoft.NET и в результате или будет обработано способом, который программист не контролирует, или возникнет сообщение о необрабатываемом исключении (Рисунок 83).

2.12.3. Метаданные и атрибуты класса

Теперь исключение обрабатывается, но проблема-то не устранена – копирование в буфер обмена до сих пор не выполняется. Внимательно прочитаем текст – следует проверить, что функция `Main` помечена атрибутом `STAThreadAttribute`. Если вспомнить, как выглядела функция `Main` сразу после создания приложения (Рисунок 39), станет понятно, что, скорее всего, речь идет об этой строчке:

[`STAThread`]

После ее добавления непосредственно перед описанием функции `Main` проблема будет устранена. Что же обозначает данная синтаксическая конструкция?

В процессе формирования исполняемого файла в нем создаются три основные структуры – код, данные и метаданные. Код – это конкретные программные инструкции, проистекающие из исходного текста программы. Данные – это области, предназначенные для размещения переменных, полей и других структур, и содержащие определенные начальные значения. Метаданные – это формальное описание структуры классов проекта, включающее в себя полный перечень классов проекта, описание каждого класса – его поля, свойства, методы и события. Например, для поля класса будет описано его имя, тип данных, комментарий, модификатор доступа. Для класса в целом метаданные содержат его полное наименование (включая пространство имен), модификатор доступа, комментарий. Метаданные реализованы в виде специальных объектов, которые относятся к пространству имен `System.Reflection`.

Кроме жестко определенной структуры метаданных, имеется возможность добавлять к ним дополнительную информацию. Эта информация может быть впоследствии проанализирована программным кодом, и поведение программы в целом может меняться в зависимости от нее. Эта Дополнительная информация называется атрибутами (класса, поля, свойства и т.п.).

Атрибуты описываются непосредственно до синтаксической конструкции, к которой они относятся. Для описания атрибутов используются квадратные скобки. Каждый атрибут имеет свое собственное имя и фактически является объектом специализированного класса. Так, в нашем случае, атрибут `STAThread` создает объект класса `System.STAThreadAttribute`. Для краткости суффикс `Attribute` в имени атрибута можно опускать. Описания `[STAThread]` и `[STAThreadAttribute]` являются эквивалентными.

Описание атрибута на самом деле является вызовом конструктора класса. Многие атрибуты имеют конструкторы без параметров (как `STAThread`), и наличие скобок в описании атрибута является в этом случае необязательным. Описания `[STAThread]` и `[STAThread()]` являются эквивалентными. Но в случае наличия у атрибута конструктора с параметрами скобки становятся обязательными, как при любом вызове метода. Примеры таких атрибутов будут рассмотрены в рамках учебного курса позже.

Особенностью атрибутов является тот факт, что их объекты конструируются только при необходимости, когда какой-либо программный код запрашивает значения атрибутов данного класса или метода. Таким образом, наличие неиспользуемых атрибутов не снижает быстродействие приложения и не повышает его требования к памяти.

В данном конкретном случае к главной функции программы `Main()` применен атрибут `[STAThread]`. Он означает, что приложение исполняется в так называемом однопоточном контейнере, что позволяет ему взаимодействовать с операционной системой, используя механизмы COM/OLE. Подробное описание этот выходит за рамки настоящих указаний, следует только отметить, что такое взаимодействие понадобилось именно для работы с буфером обмена Windows. Все остальные функции приложения не требуют наличия данного атрибута.

2.13. Целые и вещественные числа

2.13.1. Переход к вещественной арифметике

До текущего момента калькулятор обрабатывал целые числа. Имеет смысл выяснить пределы возможностей этого типа данных. Тип данных `int` полностью именуется как `System.Int32` и, в соответствии

с наименованием, использует 32 бита для хранения целого числа. Минимальное значение для данного типа данных равно $-2^{31} = -2147483648$, а максимальное значение составляет $2^{31} - 1 = 2147483647$ (Рисунок 87).

Номер бита																																Значение ₁₀			
3	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0			
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1 972
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	100 500	
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2 147 483 647	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-2 147 483 648	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1 001	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1	

Бит

Байт

Рисунок 87. Хранение целых чисел в памяти компьютера в дополнительном коде

Поэтому если в калькуляторе начать набирать число, состоящее, например, из одних единиц, то рано или поздно вместо добавления очередной цифры возникает переполнение, приводящее к некорректному результату (Рисунок 88).

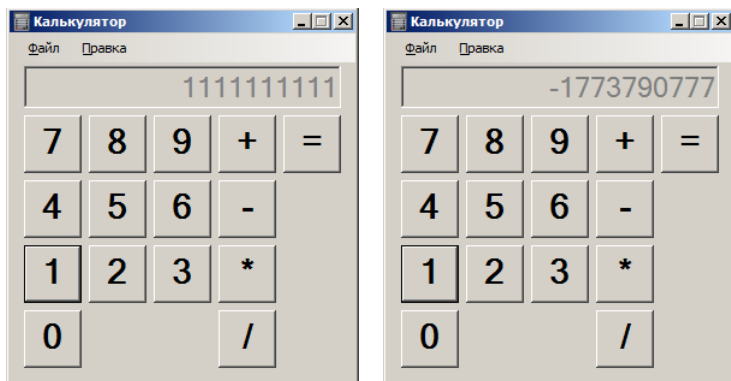


Рисунок 88. Целочисленное переполнение

Второй особенностью целочисленной арифметики является то, что деление на ноль (в отличие от целочисленного переполнения) приводит к исключению (Рисунок 89).

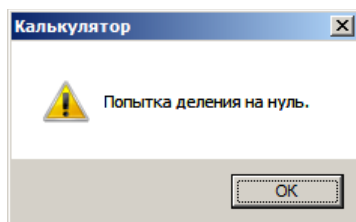


Рисунок 89. Исключение `System.DivideByZeroException`

Переход на тип данных `System.Int64` – 64-битное целое со знаком – является только полумерой, так как обе проблемы сохранятся и в этом случае. Поэтому имеет смысл рассмотреть переход от целых к вещественным числам. В языке C# имеются три основных типа данных для работы с вещественными числами – `float`, `double` и `decimal`. Полный перечень числовых типов данных приведен в разделе 5. Для целей калькулятора наиболее подходит использование типа `double`.

Для осуществления такого перехода требуется аккуратная замена типа `int` на тип `double` по всему тексту программы (конечно, где это применимо). Собственно, это требуется сделать в описании полей `_x`, `y` и свойстве `x`, а также в методе `pasteToolStripMenuItem_Click`. В последнем случае также требуется переход от использования метода `int.TryParse` к методу `double.TryParse`.

После такой модификации калькулятор становится способным обрабатывать достаточно значительные числа, при этом, если число становится слишком большим, то оно представляется в экспоненциальной (инженерной) нотации (Рисунок 90). Следует обратить внимание на то, что преобразование в строку обеспечивает всё тот же метод `ToString`. Но так как в этом случае он вызывается для вещественного, а не для целого числа, результат преобразования может быть другим.

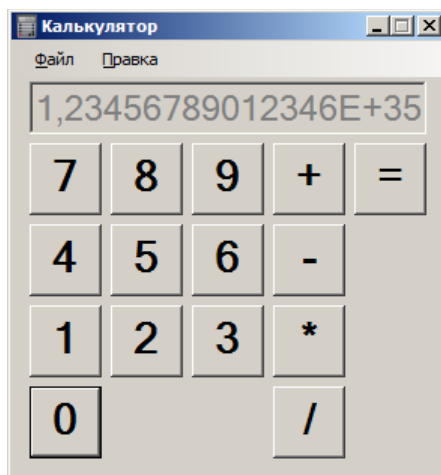


Рисунок 90. Вещественное число в инженерной нотации

Другой важной особенностью, которую обрел калькулятор, является возможность деления на ноль без возникновения исключения. В

этом случае формируется специальное значение `double.PositiveInfinity` или `double.NegativeInfinity` (в зависимости от знака выражения), которое используется для представления бесконечности (Рисунок 91).

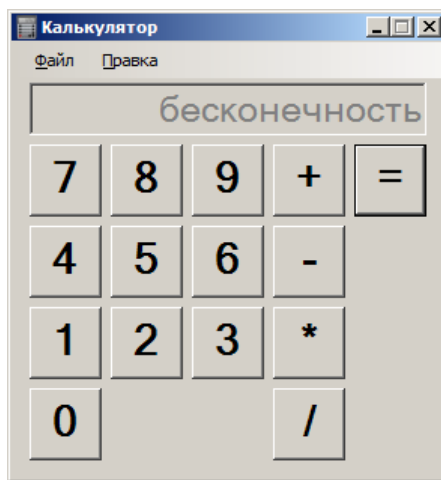


Рисунок 91. Значение `double.PositiveInfinity`

При попытке деления нуля на ноль (Рисунок 92) возникает еще одно специальное значение - `double.NaN` (Not a Number – не число). К такому же результату приведет любая некорректная арифметическая или алгебраическая операция, например, извлечение квадратного корня из отрицательного числа. Значение NaN является поглощающим значением – любая операция, в которой в качестве одного из аргументов выступает значение NaN, приведет к результату, равному NaN (в этом проявляется сходство с значением NULL языка SQL).

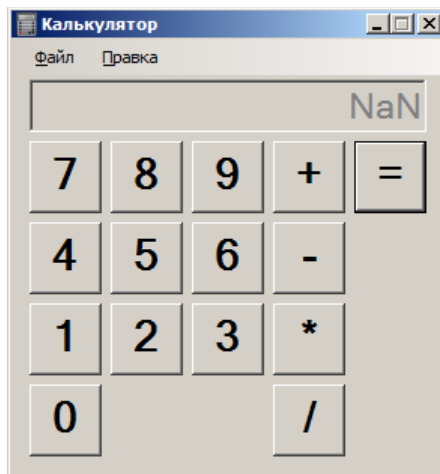


Рисунок 92. Значение `double.NaN`

Наличие в вещественной арифметике значения NaN и отсутствие исключения при делении на ноль и других подобных действиях имеет серьезное основание. В соответствии со стандартом IEEE 754-2008 выполнение любых математических операций не должно приводить к аварийному завершению программы вне зависимости от того, насколько корректны исходные данные. Поэтому любые математические операции с вещественными числами C# всегда завершаются без выбрасывания исключений. Но при этом имеет смысл выполнять проверку результата на корректность. Если в результате вычислений получилось значение NaN, это означает, что в исходных данных или формулах была допущена какая-либо ошибка.

2.13.2. Кнопка для десятичной точки

Так как калькулятор перешел на работу с вещественными числами, имеет смысл реализовать еще одну кнопку для ввода десятичной точки (традиционно в программировании используется именно десятичная точка, а не десятичная запятая). Кажущаяся простота этой задачи обманчива – надо действовать вдумчиво и по порядку. Для

начала требуется добавить в дизайн формы очередную кнопку и создать для нее отдельный обработчик. Для определенности назовем кнопку `buttonDot`. Далее следует немного подумать и определить места в программе, где потребуется внести изменения.

Для начала полезно представить себе возможный сценарий работы калькулятора. Например, пользователь нажимает последовательно клавиши `[1]`, `[.]`, `[0]`, `[1]`. Соответственно, индикатор должен последовательно же отображать строки «1», «1.», «1.0», «1.01».

Первая проблема в том, что для компьютера значения «1» и «1.» неразличимы, коль скоро мы пользуемся вещественной арифметикой. Следовательно, где-то надо иметь какой-то признак, который говорил бы о том, была ли кнопка с точкой нажата ранее или нет.

Вторая проблема состоит в том, что и значения «1» и «1.0» для компьютера совершенно неразличимы, а пользователь ожидает увидеть на экране ноль, даже если он не является значащим. Следовательно, потребуется счетчик, который бы считал количество десятичных цифр, введенным пользователем. Этот же счетчик, скорее всего, потребуется и для правильного вычисления нового значения свойства `x`. В текущий момент при нажатии очередной цифры значение вычисляется при помощи следующего оператора:

```
x = 10 * x + digit;
```

Понятно, что если мы, скажем, набираем сотые, то оператор должен выглядеть, например, так (оператор `x += a` является сокращенной формой записи для оператора `x = x + a`):

```
x += digit * 0.01;
```

Итого для начала следует описать два новых поля. Логическое поле `dotPressed`, которое будет хранить информацию о том, была ли уже нажата десятичная кнопка или нет. И целочисленное поле `factor`, которое будет уже введенное количество десятичных знаков после точки. Это может выглядеть вот так:

```
private bool dotPressed = false;
private int factor;
```

В коде обработчика нажатия на десятичную точку следует установить признак `dotPressed` в значение `true` и очистить переменную `factor` (Рисунок 93).

```
private void buttonDot_Click(object sender, EventArgs e)
{
    dotPressed = true;
    factor = 0;
}
```

Рисунок 93. Обработчик нажатия на десятичную точку

2.13.3. Модификация свойства `x`

Вспомним, где формируется значение индикатора. Это происходит в блоке `set` свойства `x`. Его потребуется существенно модифицировать. Возможный вариант выглядит следующим образом (Рисунок 94).

```
set
{
    // Сохранение значения
    _x = value;
    // Строка формата для преобразования
    // числа в строковое представление
    string format = string.Empty;
    if (dotPressed)
    {
        if (factor == 0)
```

```

    {
        format = @"0\.";
    }
    else
    {
        format = "0.";
        // Формируем строку вида 0.000 в соответствии
        // с количеством десятичных цифр
        for (int i = 0; i < factor; i++)
        {
            format += "0";
        }
    }
}
// Обновление индикатора
indicator.Text = _x.ToString(format);
}

```

Рисунок 94. Модифицированный блок set свойства x

Разберем этот код по косточкам. Для начала видно, что метод ToString теперь используется в варианте с параметром. В качестве параметра передается так называемая строка формата – специальным образом сформированная строка, которая описывает, как именно требуется представить число в строковой форме. Вначале строка формата пуста (используется системное значение `string.Empty`, что эквивалентно заданию пустой строки в форме `""`). Вызов `ToString("")` полностью эквивалентен вызову `ToString()`, который использовался ранее.

Если точка была нажата, но ни одной десятичной цифры еще не введено, используется особая строка формата `"0\."`. Ноль означает, что нулевое значение должно быть представлено как 0, а следующие два символа обеспечивают принудительное формирование символа десятичной точки. В языке C# по умолчанию используется унаследованный от языков C/C++ способ задания специальных сим-

волов при помощи обратной косой черты. В данном случае требуется сформировать именно обратную косую черту, за которой идет точка. По стандартным правилам C# такую строку следовало бы записать в виде "0\\.". Но язык предлагает альтернативный способ задания строковых констант, в которых нет никаких специальных символов. Для этого перед первой кавычкой следует поставить символ @. В данном случае записи "0\\." и @"0\." полностью эквивалентны.

Если было введено сколько-нибудь десятичных цифр, то требуется сформировать строку формата вида "0.000", где количество нулей после точки равно значению поля factor. В данном случае точка в строке формата обозначает положение десятичной точки, и использование обратной косой черты не требуется. Для формирования строки из повторяющихся символов используется стандартный цикл for, при этом счетчик цикла i объявляется прямо внутри оператора цикла. Оператор инкремента i++ увеличивает значение переменной i на единицу и возвращает значение переменной до увеличения (по этой причине он часто называется оператором постинкремента).

Наконец, осуществляется формирование строкового представления индикатора в соответствии со строкой формата, полученной выше.

2.13.4. Модификация обработчика цифровой кнопки

Очевидно, что требуется изменить и обработчик цифровой кнопки так, чтобы он учитывал поля dotPressed и factor. Возможный вариант приведен ниже. Приведен только модифицированный фрагмент обработчика (Рисунок 95).

```
// Проверка на начало ввода нового числа
if (newNumber)
{
    // Начало ввода нового значения
    x = digit;
```

```
        // Сброс признака ввода нового числа
        newNumber = false;
    }
    else if (dotPressed)
    {
        // Ввод дробной части числа
        x += digit / Math.Pow(10, ++factor);
    }
    else
    {
        // Ввод целой части числа
        x = 10 * x + digit;
    }
}
```

Рисунок 95. Модифицированный обработчик цифровых кнопок

Фактически изменения сосредоточены в одной строке, которой предшествует дополнительное условие (Рисунок 95, выделено желтым фоном). Здесь используется функция возведения в степень Pow, являющаяся статическим методом класса Math, содержащего все основные математические возможности языка C#. Так как при вводе первого дробного разряда (десятков) поле factor содержит значение ноль, то тут используется оператор преинкремента ++factor. Оператор сначала увеличивает значение поля на единицу, а потом возвращает результат этого увеличения. Также для краткости записи используется оператор +=.

2.13.5. Отладка приложения

После запуска доработанного таким образом калькулятора выясняется, что, несмотря на все усилия, после нажатия точки ничего на экране не происходит, хотя далее при вводе любой цифры (включая ноль) все работает корректно. В чем же дело? Проблема в том, что значение свойства x при нажатии на точку не меняется, и код свойства не вызывается. Исправить это можно двумя способами. Первый

способ очевидный – добавить в код обработчика нажатия на точку строку вида:

```
indicator.Text += ".";
```

Проблема будет устранена, но с точки зрения архитектуры приложения это некорректно. Смысл написания именно свойства (а не поля) `x` был именно в том, что в нем инкапсулируется вся работа с текстовым представлением индикатора, и нигде больше индикатор в явном виде не присваивается. Кроме того, такой вариант решения делает ненужным ветку свойства `if (factor == 0)`.

Более изящным вариантом является следующая строка:

```
x = x;
```

Несмотря на кажущуюся бессмысленность, этот вариант достигает цели, так как блок `set` свойства `x` в данном случае вызывается принудительно, а так как его выполнение, в том числе, зависит и от значений полей `dotPressed` и `factor`, то результат вызова будет соответствовать целям. Следует только добавить эту строку именно в конец обработчика (Рисунок 96), а не в его начало.

Замечание. Описанный способ реализации не является, естественно, ни единственным, ни единственно верным. Он намеренно выбран для выпуклой иллюстрации возможностей языка C#, в частности, для осознания отличия полей и свойств. Альтернативным вариантом реализации мог бы быть специальный метод, который бы заполнял значение свойства `indicator.Text` и который можно было бы вызывать по мере необходимости.

Второй проблемой калькулятора является то, что после перехода к вводу нового числа (после нажатия на кнопку арифметической опе-

рации) флаг нажатия десятичной точки не очищается. Везде, где идет присваивание `newNumber = true`, требуется добавить строку `dotPressed = false`. Тут возникает желание вынести эти две строки в отдельный метод— имеет смысл подумать об этом или как минимум запомнить этот факт.

Наконец, еще одной ошибкой является то, что десятичную точку можно нажать несколько раз. Самым простым способом устранить ее является введение дополнительной проверки (Рисунок 96). Для проверки используется условный оператор и логическая операция отрицания НЕ, которая в языке C# записывается в виде восклицательного знака.

```
private void buttonDot_Click(object sender, EventArgs e)
{
    if (!dotPressed)
    {
        dotPressed = true;
        factor = 0;
        x = x;
    }
}
```

Рисунок 96. Исправленный обработчик нажатия на десятичную точку

2.13.6. Инвариантные региональные настройки

Если в региональных настройках операционной системы в качестве разделителя целой и дробной части числа выбрана запятая, то именно этот символ будет использован методом `ToString` для представления дробных чисел. При этом незначащая десятичная точка остается точкой, так как она запрограммирована жестко (Рисунок 97).

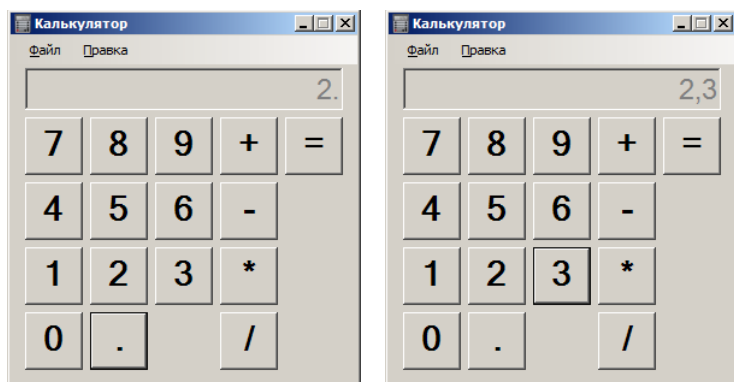


Рисунок 97. Влияние региональных настроек на результат вызова ToString

Понятно, что профессиональная программа так себя вести не должна. Есть два варианта устранения этого нюанса – переход к инвариантным региональным настройкам или учет региональных настроек в коде программы.

Инвариантные региональные настройки используются тогда, когда выполнение программы не должно никак зависеть от того, что содержат региональные настройки конкретной операционной системы. Для их использования в данном случае метод ToString должен быть вызван с дополнительным параметром:

```
indicator.Text = _x.ToString  
    (format, System.Globalization.CultureInfo.InvariantCulture);
```

Данный параметр задает региональные настройки, которые используются при преобразовании числа в строку. В качестве значения параметра используется статическое свойство InvariantCulture класса CultureInfo. Оно возвращает унифицированный набор настроек, не зависящих от конкретного региона. В этом наборе, в частности, в качестве разделителя целой и дробной части числа используется символ точки.

Кроме того, соответствующим образом следует изменить обработчик пункта меню «Вставка» - метод TryParse должен быть вызван с инвариантными региональными настройками:

```
if (double.TryParse(Clipboard.GetText(),
    System.Globalization.NumberStyles.Float,
    System.Globalization.CultureInfo.InvariantCulture, out value))
```

2.13.7. Региональные настройки операционной системы

Более интересен вариант учета региональных настроек, выполненных на уровне операционной системы. Такой подход позволяет разрабатывать приложения, которые подстраиваются под общепринятые в данном регионе параметры, что увеличивает дружелюбность интерфейса. Из всего набора региональных настроек потребуется только символ разделителя целой и дробной части числа. Разумно определить его однократно, в момент загрузки программы. Для хранения этого символа понадобится очередное поле, на этот раз строковое:

```
private string dot;
```

Для его инициализации в конструктор формы добавляется еще один оператор присваивания, который использует сведения о текущих региональных настройках – свойство CurrentCulture:

```
dot = System.Globalization.CultureInfo.CurrentCulture.
    NumberFormat.CurrencyDecimalSeparator;
```

Далее имеет смысл сделать так, чтобы этот символ отображался и на соответствующей кнопке калькулятора:

```
buttonDot.Text = dot;
```

Наконец, следует модифицировать блок set свойства x в той его части, где символ десятичной точки задан в виде константы:

```
if (factor == 0)
```

```
{  
    format = @"0\" + dot;  
}
```

После этого приложение корректно и автоматически настраивается на использование текущих региональных настроек в момент запуска.

2.14. Дополнительные задания

В качестве дополнительных (домашних) заданий предлагается реализовать следующие функции и возможности:

- Добавить кнопку, которая вычисляет квадратный корень (унарная операция).
- Добавить кнопку, которая меняет знак числа (унарная операция).
- Добавить кнопку для возведения числа в произвольную степень (бинарная операция).
- Добавить кнопку [C] - полный сброс калькулятора.
- Добавить переключатель систем счисления и обработку чисел в различных системах счисления. Следует реализовать поддержку систем счисления с основанием в диапазоне от 2 до 16 (Рисунок 98).
- Добавить кнопки работы с памятью (M+, M-, MR, MC).

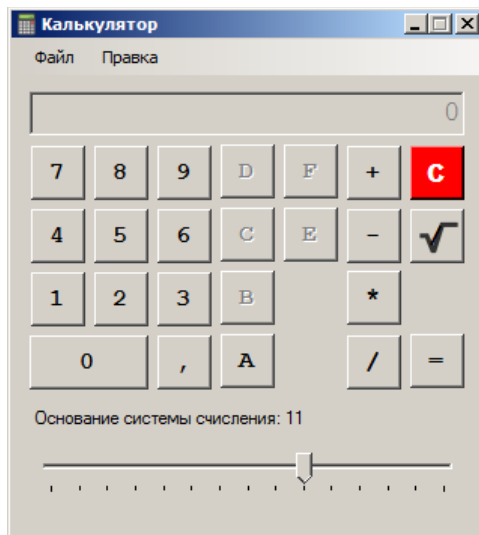


Рисунок 98. Пример интерфейса с дополнительными возможностями

3. Терминология языка C#

Таблица 2. Термины языка C#

Термин	Определение
IDE	Интегрированная среда разработки (Integrated Development Environment). Приложение, которое предоставляет универсальный пользовательский интерфейс для различных средств разработки, в том числе компилятора, отладчика, редактора кода и конструкторов.
Анонимный метод	Анонимный метод — это блок кода, который передается в качестве параметра делегату.
Базовый класс	Класс, от которого наследует другой, "производный" класс.

Термин	Определение
Вложенный тип	Тип, объявленный внутри объявления другого типа.
Делегат	Делегат — это тип, который ссылается на метод. Когда делегату назначается методу, он ведет себя в точности, как этот метод.
Деструктор	Особый метод класса или структуры, который подготавливает экземпляр для уничтожения системой.
Доступный член	Член, доступ к которому можно получить из данного типа. Член, доступный для одного типа, не обязательно доступен для другого типа.
Изменяемый тип	Тип, после создания экземпляра которого свойства, поля и данные этого экземпляра могут изменяться. Изменяемыми являются почти все ссылочные типы.
Интерфейс	Тип, который содержит только подписи открытых методов, событий и делегатов. Объект, который наследует от интерфейса, должен реализовывать все методы и события, определенные в интерфейсе. Классы и структуры могут наследовать любое количество интерфейсов.
Итератор	Итератором называют метод, который позволяет объектам-получателям класса, содержащего коллекцию или массив, использовать оператор foreach для перебора коллекции или массива.
Класс	Тип данных, описывающий объект. Классы содержат данные и методы обработки этих данных

Термин	Определение
Конструктор	Особый метод класса или структуры, который инициализирует объект данного типа.
Метод	Именованный блок кода, который предоставляет поведение класса или структуры.
Метод доступа	Метод, который устанавливает или извлекает значение закрытого члена данных, связанное со свойством. Для свойств, доступных для чтения и записи, предусмотрены методы доступа get и set . Для свойств, доступных только для чтения, применяется только метод доступа get .
Модификатор доступа	Зарезервированное слово, например private , protected , internal или public , которое ограничивает доступ к типу или члену типа.
Наследование	Язык C# поддерживает наследование. Это означает, что класс, производный от другого класса, называемого базовым, наследует те же методы и свойства. В наследовании участвуют базовые и производные классы.
Недоступный член	Член, доступ к которому невозможно получить из данного типа. Член, недоступный для одного типа, не обязательно недоступен для другого типа.
Неизменяемый тип	Тип, после создания экземпляра которого свойства, поля и данные этого экземпляра не изменяются. Большинство типов значений являются неизменяемыми.
Объект	Экземпляр класса. Объект существует в памяти и содержит данные и методы для обработки этих данных.

Термин	Определение
Оптимизация кода	Повторное использование ранее введенного кода. Редактор кода может выполнить интеллектуальное форматирование кода, чтобы, например, преобразовать выделенный блок кода в метод.
Поле	Член данных класса или структуры, к которому можно получить непосредственный доступ.
Производный класс	Класс, который использует наследование для получения, расширения или изменения данных другого, "базового" класса.
Свойство	Член данных, доступ к которому осуществляется посредством метода доступа.
Событие	Член класса или структуры, который отправляет уведомления об изменении.
Ссылочный тип	Тип данных. Переменная, объявленная как ссылочный тип, указывает на расположение, в котором хранятся данные.
Статический	Для существования класса или метода, объявленного статическим, не требуется создавать его экземпляр с помощью ключевого слова new . В качестве примера статического метода можно назвать метод <code>Main()</code> .
Стек вызова	Ряд вызовов метода, который начинается с запуска программы и заканчивается оператором, выполняющимся в данный момент.

Термин	Определение
Структура	Составной тип данных, содержащий, как правило, несколько переменных, между которыми установлено некоторое логическое отношение. Структуры могут также содержать методы и события. Структуры не поддерживают наследование, но поддерживают интерфейсы. Структура является типом значения, тогда как класс — это ссылочный тип.
Тип значения	Тип значения — это тип данных, который располагается в стеке, в отличие от ссылочного типа, располагающегося в куче. Типами значения являются встроенные типы, в том числе числовые типы, а также тип структуры и тип "nullable". Ссылочными типами являются тип класса и строковый тип.
Универсальные шаблоны	Универсальные шаблоны позволяют определить класс и метод, который определяется с помощью параметра типа. Когда клиентский код создает экземпляр типа, в качестве аргумента он указывает определенный параметр.
Член	Поле, свойство, метод или событие, объявленное в классе или структуре.

4. Зарезервированные ключевые слова C#

Ключевые слова — это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения для компилятора. Их нельзя использовать в программе в качестве идентификаторов, если только они не содержат префикс @. Например,

@if является допустимым идентификатором, но if таковым не является, поскольку if – это ключевое слово.

В частности, ключевые слова нельзя использовать как имена классов, пространств имен. А так как при создании проекта его имя становится именем его пространства имен по умолчанию, то и в качестве имени проекта ключевые слова лучше не использовать.

4.1. Основные ключевые слова

Ниже в таблице приведен список основных ключевых слов.

Таблица 3. Основные ключевые слова языка C#

Ключевое слово	Краткое описание
abstract	Абстрактный класс или член класса Абстрактный класс не имеет объектов этого класса Абстрактный метод класса не имеет реализации
as	Используется для приведения типов
base	Базовый (класс, метод)
bool	Логический тип данных. Возможные значения: false , true
break	Выход из цикла
byte	Целочисленный тип данных без знака в интервале от 0 до $2^8 - 1$ 255
case	Вариант ветвления в операторе switch
catch	Ловушка для исключения
char	Символьный тип данных – одна буква
checked	С проверкой
class	Класс
const	Константа
continue	Переход к проверке условия цикла
decimal	Десятичный тип данных

Ключевое слово	Краткое описание
default	Вариант ветвления по умолчанию в операторе switch
delegate	Делегат (метод, передаваемый в качестве параметра)
do	Цикл «после»
double	Вещественный тип данных с двойной точностью
else	Ветка условного оператора
enum	Перечислимый тип данных
event	Событие
explicit	Явное преобразование типа
extern	Внешний объект
false	Логическая константа «ложь»
finally	Блок кода, выполняющийся в самом конце вне зависимости от наличия исключений
fixed	Фиксация адреса
Float	Вещественный тип данных с одинарной точностью
for	Цикл со счетчиком
foreach	Цикл-итератор по всем элементам перечислимого объекта
goto	Оператор безусловного перехода к заданной метке (о ужас, он всё ещё существует)
if	Условный оператор
implicit	Неявное преобразование типа
in	Часть конструкции цикла foreach
int	Целочисленный тип данных со знаком
inter- face	Интерфейс
internal	Внутренний
is	Проверка на соответствие заданному типу
lock	Блокировка одновременного (многопоточного) выполнения. Вынуждает фрагмент кода выполняться в единственном потоке, запрещает многозадачность на этом

Ключевое слово	Краткое описание
	участке
long	Целочисленный тип данных со знаком
namespace	Пространство имен
new	Создание нового объекта
null	Пустая ссылка, отсутствие значения
object	Объект – корневой класс в иерархии объектов .NET
operator	Оператор
out	Изменяемый параметр
override	Перекрытие метода
params	Параметры
private	Модификатор области видимости. Минимальная область видимости – только внутри класса
protected	Модификатор области видимости. Видимость только внутри класса и его потомков.
public	Модификатор области видимости. Полная видимость.
readonly	Член класса только для чтения (инициализация только внутри конструктора)
ref	Передача по ссылке
return	Возврат значения из метода
sbyte	Целочисленный тип данных со знаком
sealed	Класс, не имеющий наследников
short	Целочисленный тип данных со знаком
sizeof	Размер объекта
stack-alloc	Распределение стека в небезопасном участке кода
static	Статический элемент – относящийся к классу в целом, не имеющий контекста объекта класса
string	Строковый тип данных
struct	Структура

Ключевое слово	Краткое описание
switch	Оператор-переключатель
this	Ссылка на текущий экземпляр класса, используется также для объявления индексатора класса
throw	Выброс (генерация, формирование) исключения
true	Логическая константа «истина»
try	Проверка на наличие исключения
typeof	Определение типа объекта
uint	Целочисленный тип данных без знака
ulong	Целочисленный тип данных без знака
unchecked	Без проверки
unsafe	Небезопасно
ushort	Целочисленный тип данных без знака
using	Использование пространства имен (директива) Определение области действия объекта, после которого он должен быть уничтожен, а используемые им ресурсы освобождены (оператор)
virtual	Виртуальный (перекрываемый) метод
void	Нет значения
volatile	Поле, которое может быть непредсказуемо изменено в многопоточной среде. Исключается из оптимизации, которая предполагает однопоточное выполнение кода
while	Цикл «пока»

4.2. Контекстные ключевые слова

Контекстные ключевые слова были введены в язык C# в процессе его развития. Контекстные ключевые слова не являются жестко зарезервированными, но использовать их в качестве идентификаторов не рекомендуется.

Таблица 4. Контекстные ключевые слова языка C#

Ключевое слово	Краткое описание
add	Обработчик подписки на событие
alias	Псевдоним для одновременного подключения библиотек с идентичным полным именем, но различными версиями
ascending	По возрастанию
async	Асинхронная операция
await	Ожидание завершения асинхронной операции
descending	По убыванию
dynamic	Объявление переменной как динамической предотвращает проверку типов на этапе компиляции программы. Все проверки осуществляются в момент выполнения программы
from	Элемент запроса LINQ
get	Описание метода чтения значения свойства
global	Ссылка на глобальное (безымянное) пространство имен
Group	Элемент запроса LINQ. Определяет группировку записей
into	Элемент запроса LINQ. Используется для построения сложных запросов
join	Элемент запроса LINQ. Используется для выборки данных из нескольких таблиц
let	Элемент запроса LINQ.
orderby	Элемент запроса LINQ. Указывает на порядок сортировки записей

Ключевое слово	Краткое описание
partial	Частичный класс или метод. Частичный класс (интерфейс, структура) размещается в двух или более файлах. Частичный метод может быть определен (при помощи заголовка) в одном из таких файлов, а реализован в другом файле.
remove	Обработчик удаления подписки на событие
select	Элемент запроса LINQ
set	Описание метода записи значения свойства
value	В методе записи значения свойства используется как неявный входной параметр метода – записываемое значение
var	Описание переменной без указания ее типа. Тип переменной определяется по типу выражения, при помощи которого она инициализируется
where	Элемент запроса LINQ, определяющий условие выборки данных
yield	Используется в методах (свойствах, операторах), которые возвращают итераторы. Определяет очередной элемент данных, возвращаемый итератором.

5. Скалярные типы данных

Для целочисленных типов данных (Таблица 5) приведены количество бит в двоичном представлении числа, наличие знака и диапазон возможных значений (для наглядности в качестве дополнительного разделителя троек цифр использована запятая).

Для вещественных типов данных (Таблица 6) приведены количество бит в двоичном представлении числа, количество значащих цифр без потери точности и диапазон возможных значений.

Таблица 5. Целочисленные типы данных

Тип C#	.NET Framework	Бит	Знак	Диапазон
sbyte	System.SByte	8	Да	-128 ... 127
byte	System.Byte	8	Нет	0 ... 255
short	System.Int16	16	Да	-32,768 ... 32,767
ushort	System.UInt16	16	Нет	0 ... 65,535
int	System.Int32	32	Да	-2,147,483,648 ... 2,147,483,647
uint	System.UInt32	32	Нет	0 ... 4,294,967,295
long	System.Int64	64	Да	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807
ulong	System.UInt64	64	Нет	0 ... 18,446,744,073,709,551,615

Таблица 6. Вещественные типы данных

Тип C#	.NET Framework	Бит	Цифры	Диапазон
float	System.Single	32	7	$\pm 1.5 \cdot 10^{-45} \dots \pm 3.4 \cdot 10^{38}$
double	System.Double	64	15-16	$\pm 5.0 \cdot 10^{-324} \dots \pm 1.7 \cdot 10^{308}$
decimal	System.Decimal	128	28-29	$\pm 7.9 \cdot 10^{-28} \dots \pm 7.9 \cdot 10^{28}$

ЛИТЕРАТУРА

C# 4.0. Полное руководство. Пер. с англ. / Герберт Шилдт. М.: Вильямс, 2015, 1056 с. ISBN 978-5-8459-1684-6

Язык программирования C# 5.0 и платформа .NET 4.5. Пер. с англ. / Эндрю Троелсен, М.: Вильямс, 2015, 1312 с. ISBN 978-5-8459-1957-1

Оглавление

Введение	3
1. Система контроля версий TFS.....	3
1.1. Общие сведения.....	3
1.2. Подключение к portalу.....	4
1.3. Настройка среды разработки.....	9
1.4. Учетные данные при подключении к TFS.....	17
1.5. Использование	23
1.6. Возможные проблемы при сборке проекта.....	25
1.6.1. Системная сборка не может быть загружена.....	25
2. Разработка простого калькулятора на базе WinForms.....	27
2.1. Создание проекта.....	27
2.2. Свойства проекта.....	30
2.2.1. Свойства приложения	32
2.2.2. Остальные разделы свойств проекта.....	42
2.3. Главная функция программы.....	42
2.4. Главная форма.....	46
2.5. Добавление первых элементов управления.....	49
2.5.1. Рекомендации по именованию идентификаторов.....	51
2.5.2. Настройка текстового поля.....	53
2.5.3. Настройка кнопки.....	54
2.6. Обработка событий элементов управления.....	55
2.6.1. Класс и конструктор класса	56

2.6.2. Описание класса.....	57
2.6.3. Модификаторы доступа	58
2.6.4. О пользе комментариев.....	60
2.6.5. Код обработчика события.....	62
2.7. Добавление цифровых элементов управления.....	64
2.7.1. Порядок обхода элементов управления (Tab Order)	65
2.8. Универсальный обработчик цифровой кнопки.....	68
2.9. Первое действие: сложение двух чисел	74
2.9.1. Обработчик операции сложения.....	75
2.9.2. Модификация обработчика цифровых клавиш.....	76
2.9.3. Обработчик кнопки вычисления	77
2.9.4. Поля и свойства	78
2.10. Остальные арифметические действия.....	82
2.10.1. Перечислимые типы данных.....	83
2.10.2. Преобразование из строки в перечислимый тип.....	86
2.10.3. Оператор множественного выбора.....	88
2.11. Добавление главного меню формы.....	93
2.12. Обработка событий меню формы.....	97
2.12.1. Буфер обмена Windows.....	97
2.12.2. Исключение и обработка ошибок.....	99
2.12.3. Метаданные и атрибуты класса.....	102
2.13. Целые и вещественные числа.....	104
2.13.1. Переход к вещественной арифметике.....	104

2.13.2. Кнопка для десятичной точки.....	109
2.13.3. Модификация свойства x.....	111
2.13.4. Модификация обработчика цифровой кнопки.....	113
2.13.5. Отладка приложения.....	114
2.13.6. Инвариантные региональные настройки.....	116
2.13.7. Региональные настройки операционной системы.....	118
2.14. Дополнительные задания.....	119
3. Терминология языка C#	120
4. Зарезервированные ключевые слова C#.....	124
4.1. Основные ключевые слова.....	125
4.2. Контекстные ключевые слова.....	128
5. Скалярные типы данных	130
Литература	131