# Урок №1

## Содержание

1. Введение в платформу Microsoft .NET
   a. История и этапы развития технологий программирования
   b. Причины возникновения платформы Microsoft .NET
   c. Сравнительный анализ преимуществ и недостатков платформы Microsoft .NET
2. Базовые понятия платформы Microsoft .NET
   a. Архитектура платформы Microsoft .NET
   b. Общеязыковая среда исполнения CLR (common language runtime)
   c. Стандартная система типов CTS (common type system)
   d. Стандартная языковая спецификация CLS (common language specification)
   e. Библиотека классов FCL (BCL)
   f. Языки платформы Microsoft .NET
   g. Схема компиляции и исполнения приложения платформы Microsoft .NET
   h. Язык MSIL (Microsoft Intermediate Language)
   i. Понятия метаданных, манифеста, сборки
3. Введение в язык программирования C#
   a. Простейшая программа на языке программирования C#
4. Рефлекторы и дотфускаторы
   a. Что такое рефлектор?
   b. Необходимость использования рефлектора
   c. Что такое дотфускатор?
   d. Необходимость использования дотфускаторов
   e. Обзор существующих дотфускаторов
5. Типы данных
   a. Целочисленные типы данных
   b. Типы данных для чисел с плавающей точкой
   c. Символьный тип данных
   d. Другие типы данных
6. Литералы
7. Переменные
   a. Понятие переменной
   b. Правила именования переменных
   c. Область видимости переменных
8. Ввод, вывод в консольном приложении
9. Структурные и ссылочные типы
10. Преобразование типов
    a. Явное преобразование
    b. Неявное преобразование

11. Операторы
    a. Арифметические операторы
    b. Операторы отношений
    c. Логические операторы
    d. Битовые операторы
    e. Оператор присваивания
    f. Приоритет операторов
12. Условия
    a. Условный оператор if
    b. Условный оператор if else
    c. Условный оператор switch
    d. Оператор ?:
13. Циклы
    a. Цикл for
    b. Цикл while
    c. Цикл do while
    d. Цикл foreach
    e. Инструкция break
    f. Инструкция continue
    g. Инструкция goto
14. Домашнее задание.

# 1. Введение в платформу Microsoft .NET

### История и этапы развития технологий программирования

То, что программные продукты, на сегодняшний день, интегрированы во все сферы деятельности человека, является общепризнанным фактом. Технологии автоматизации производственных процессов внедряются не только на промышленных объектах, но, например, и в фермерском хозяйстве.

В связи со столь бурным ростом спроса на программное обеспечение самых различных сфер жизни, получает своё развития и сама технология создания этого программного обеспечения. Которая состоит, с одной стороны, в совершенствовании языков программирования, позволяющих реализовывать всё более сложные и

масштабные проекты, а, с другой стороны, – в развитии технологий, которые либо на этих языках основаны, либо позволяют реализовать возможность создания и использования сложных высокоуровневых языков программирования.

Мы рассмотрим развитие языков программирования на примере семейства языков «C».

Первым языком этого семейства был язык программирования «C», разработанный Дэнисом Ритчи в 1970-х годах. Как и все популярные языки программирования, язык «C» произошёл из кризиса программного обеспечения, реализовав новаторский подход своего времени – «структурное программирование».

Как правило, потребность в разработке новых языков заключается в необходимости новых средств масштабирования программных решений и самого программного кода. То есть в создании таких механизмов, которые позволяли бы с лёгкостью расширять существующие возможности программы и привносить в программу новые функциональные модули. Поскольку, зачастую, основная работа над проектом происходит не на этапе проектирования и разработки, а во время «развития» и «сопровождения» (модернизации и расширения программного решения уже после его появления на рынке).

До языка «C» программирование в основном было императивным, что вызывало сложности при увеличении «размера» («масштабы») программных проектов. По-правде сказать, язык «C», хоть и решал некоторые проблемы связанные с масштабированием кода (привносил такие элементы: такие, как макроопределения, структуры и т.д.), но всё ещё имел серьёзный недостаток – невозможность справляться с большими проектами.

Следующим этапом развития семейства языков «C» стал язык «C++», разработанный Бьярни Страуструпом в 1979-ом году, который реализовывал парадигму объектно-ориентированного подхода в программировании.

Язык «C» имел большой успех, в связи с тем, что в нём сочетались гибкость, мощность и удобство. Поэтому новый язык «C++» стал «развитием» языка «C». По правде

сказать, «C++» - это объектно-ориентированный «C», и причиной его возникновения стала «мода» на объектно-ориентированное программирование.

Тесное родство «C++» и «C» обеспечило популярность новому языку программирования, поскольку «C»-программисту не нужно было изучать новый язык программирования, достаточно было освоить «новые» - объектно-ориентированные возможности и без того удачного языка.

Но время предъявило новые требования в области разработки программного обеспечения. Они заключались в том, что у конечного потребителя возникала необходимость в межплатформенной переносимости программного обеспечения, упрощения передачи проектов по сетям коммуникации, а так же в уменьшении времени, которое затрачивается на разработку программного обеспечения. Названная проблема, к сожалению, не решается созданием нового языка программирования. Она находиться в области технологии, поэтому для её решения необходима была новая технология, которая смогла бы работать одинаково эффективно на всех платформах (Windows, Unix, Linux, Mac OS), обеспечивала бы отсутствие конфликтов с операционной системой при переносе приложения с одной операционной системы на другую. Естественно, для создания новой технологии необходим был новый язык программирования, который, с одной стороны, был бы языком реализации данной технологии, а, с другой, - обеспечивал бы гибкость и скорость разработки проектов на этой новой технологии.

В 1991 году компания Sun Microsystems предложила решение этой проблемы на базу своего нового языка «Oak» (англ. Дуб), который впоследствии стал называться «Java». Авторство этого языка приписывают Джеймсу Гослингу. На базе языка Java была создана среда исполнении «Java Runtime». Межплатформенная переносимость обеспечивалась за счёт существования интегрированной среды исполнения Java. Которая могла исполнять приложения созданные на языке «Java» на любых платформах, на которых она установлена. Ограничение заключалось лишь в одном – существовании такой среды исполнения для всех существующих операционных систем. Компания Sun разработала

варианты такой среды исполнения практически для всех существующих операционных систем. На сегодняшний момент в области межплатформенной переносимости с Java-приложениями не может конкурировать ни одна технология.

Однако, язык Java решал далеко не все проблемы (например проблему межъязыкового взаимодействия). Так же, приложения созданные на Java исполняются достаточно медленно, что не позволяет использовать их на малопроизводительных платформах. И сам язык Java не содержит всех современных языковых средств и механизмов, которые предлагает C#. Но это скорее из-за того, что между их появлением 9 лет разницы, а в мире информационных технологий этот период равносилен нескольким поколениям.

Язык C# появился в 2000 году и стал основой новой стратегии развития компании Microsoft. Главным архитектором этого языка является Андерс Хейлсберг (он же в 1980-х годах был автором Turbo Pascal).

C# стал потомком языков C, C++ и Java. Можно даже сказать, что он стал их эволюционным продолжением, объединив в себе основные преимущества и доработав положительные стороны описанных языков. Поскольку язык C# позиционируется в семействе C-языков, он перенял основу синтаксиса языка C++. Поэтому C++ программисту будет достаточно легко «перейти» на C# (к слову сказать, преемственность новых языков программирования – один из принципов стратегии компании Microsoft). Необходимо отметить, что язык C# является частью платформы, называемой **платформой .NET**. Справедливости ради необходимо отметить, что C# не единственный язык, используемый в рамках **платформы .NET**, которая представляет собой, с одной стороны, технологию, а с другой – концепцию развития средств разработки программного обеспечения, позиционируемую компанией **Microsoft**. Эта технология обычно отождествляется с **Microsoft .NET Framework**, представляющим собой совокупность программных модулей (все модули будут подробно описаны в соответствующих разделах этого урока), в рамках и посредством которых в операционной системе Windows выполняются **.NET-приложения**.

**Microsoft .NET Framework** – это инсталляционный пакет, который можно свободно загрузить с сайта http://www.microsoft.com/downloads/. Тогда как **платформа .NET** – это концепция развития, принятая компанией Microsoft.

Язык C#, в свою очередь, позиционируется как связующий язык в рамках платформы **Microsoft .NET**.

## Причины возникновения платформы Microsoft .NET

Среди основных причин возникновения технологии .NET можно выделить следующие:

- **необходимость межплатформенной переносимости:** Глобализация с одной стороны и развитие технологий коммуникации – с другой, обусловили необходимость создания таких приложений, которые могли бы выполняться независимо от архитектуры операционной системы и вычислительной техники, на которой программное решение будет запускаться.
- **необходимость в упрощении процесса развёртывания** программного решения и уменьшения вероятности конфликта версий, которые с течением времени появляются всё быстрее и быстрее.
- **Необходимость создания среды исполнения программных решений**, которая бы могла предоставить режим безопасного исполнения потенциально опасных программных продуктов, решала проблемы производительности операционной системы, посредством контроля за распределением ресурсов.
- Необходимость создания технологии разработки программных решений, которая реализовывала бы все **коммуникационные возможности** в соответствии с современными промышленными стандартами и гарантировала интегрирование созданного на базе неё кода с кодом, созданным с использованием других технологий (**межъязыковая интеграция**).

## Сравнительный анализ преимуществ и недостатков платформы Microsoft .NET

Причины, или, скорее, проблемы, приводящие к возникновению технологий, не всегда полностью решаются новой технологией.

Платформа .NET эффективно решает вопрос межъязыкового взаимодействия, поскольку поддерживает механизмы, позволяющие импортировать программные модули из сборок, «написанных» на других языках, а так же приводить данные «неизвестных» типов к соответствующим типам общей системы типов **.NET Framework**. А также содержит другие программные средства, направленные на реализацию межъязыкового взаимодействия, которые будут описаны в соответствующих уроках.

Так же платформа Microsoft .NET решает проблему развёртывания приложения, реализуя архитектурную независимость приложений: Microsoft .NET приложения компилируются не в исполняемый код, а в промежуточный код (**Microsoft Intermediate Language** – **MISL** или просто **IL**), а только при запуске, посредством JIT компиляции (**Just In Time Compilation**), компилируются в исполняемый код, с учётом особенностей архитектуры той ЭВМ, на которой происходит компиляция.

Также .NET-приложения имеют сравнительно малый объём. Это происходит потому, что все необходимые для функционирования приложения средства Базовой Системы Типов (базовая библиотека классов) содержаться на каждом клиентском компьютере, (наличие установленного на клиентском компьютере .**NET Framework** является обязательным условием). Необходимые приложению модули Базовой Системы Типов не включаются в его состав (исходный код), а подключаются вовремя запуска, что уменьшает объём .NET-приложения.

Все вышесказанное упрощает передачу приложения по системам коммуникации и его дальнейшее развёртывание, даже если рабочие станции распределены на больших расстояниях друг от друга.

Одним из самых весомых достижений платформы **.NET Framework** является создание интегрированной среды времени выполнения приложений (**Common Language Runtime**), благодаря которой стало возможным использование **безопасного** (или **управляемого**) кода (**Managed Code**). Одно из преимуществ безопасного кода состоит в том, что среда исполнения сама управляет выделением памяти и различного рода ресурсов, а так же руководит доступом к ним. Это позволяет разработчику, с одной стороны, избежать утечки памяти, а, с другой, сконцентрироваться на концептуальной части кода, обособившись от вопросов, связанных с управлением памятью и ресурсами, в самом широком смысле.

Однако, наряду с преимуществами, которые нам даёт платформа .NET, она имеет и несколько серьёзных недостатков:

Во-первых, из-за того, что компиляция происходит после запуска приложения, мы получаем задержки при первом запуске приложения. Этот недостаток является весь относительным: поскольку получившийся в результате компиляции исполняемый файл сохраняется во временной директории, то при последующих запусках приложения не будет происходить повторной компиляции. Однако, если будет запущена новая версия приложения (например если изменить что-то в сборке и запустить), то произойдёт перекомпиляция приложения и старый исполняемый файл будет заменён на новый. Так же повторная компиляция будет происходить всякий раз, как получившийся временный исполняемый файл будет удалён (например: в результате действия служебных приложений, которые удаляют временные файлы).

Второй недостаток – это относительно медленное исполнение приложения вследствие того, что интегрированная среда исполнения забирает часть системных ресурсов на свои нужды (например: часть процессорного времени), а так же преломляет

все обращения приложения: приложение обращается к среде исполнения, которая, в свою очередь, получает безопасную ссылку на необходимый ресурс и возвращает его приложению, что работает медленнее, чем прямое обращение приложения к ресурсу. Однако, противовесом этому недостатку служит безопасность, которую обеспечивает среда исполнения.

Последний недостаток, который мы отметим, – это межплатформенная непереносимость .NET-приложений. Теоретически, при наличии в операционной системе .NET Framework, .NET-приложение должно корректно работать в любой операционной системе, однако .NET Framework существует только для операционных систем Microsoft Windows. Существует аналог Framework для операционных систем Linux (проект Mono), однако в силу концептуальных отличий архитектуры Linux от Windows, приложение придётся модифицировать с учётом этих отличий.

Отсутствие межплатформенной переносимости является наиболее существенным недостатком платформа Microsoft .NET, поскольку недостатки, связанные с расходом ресурсов, решаются наращиванием системных ресурсов, которые на сегодняшний день перестали составлять проблему, а недостатки, связанные с отсутствием программных средств, решаются подключением компонентов, разработанных на «альтернативных» языка программирования, в которых эти средства имеются.

## 2. Базовые понятия платформы Microsoft .NET

### Архитектура платформы Microsoft .NET.

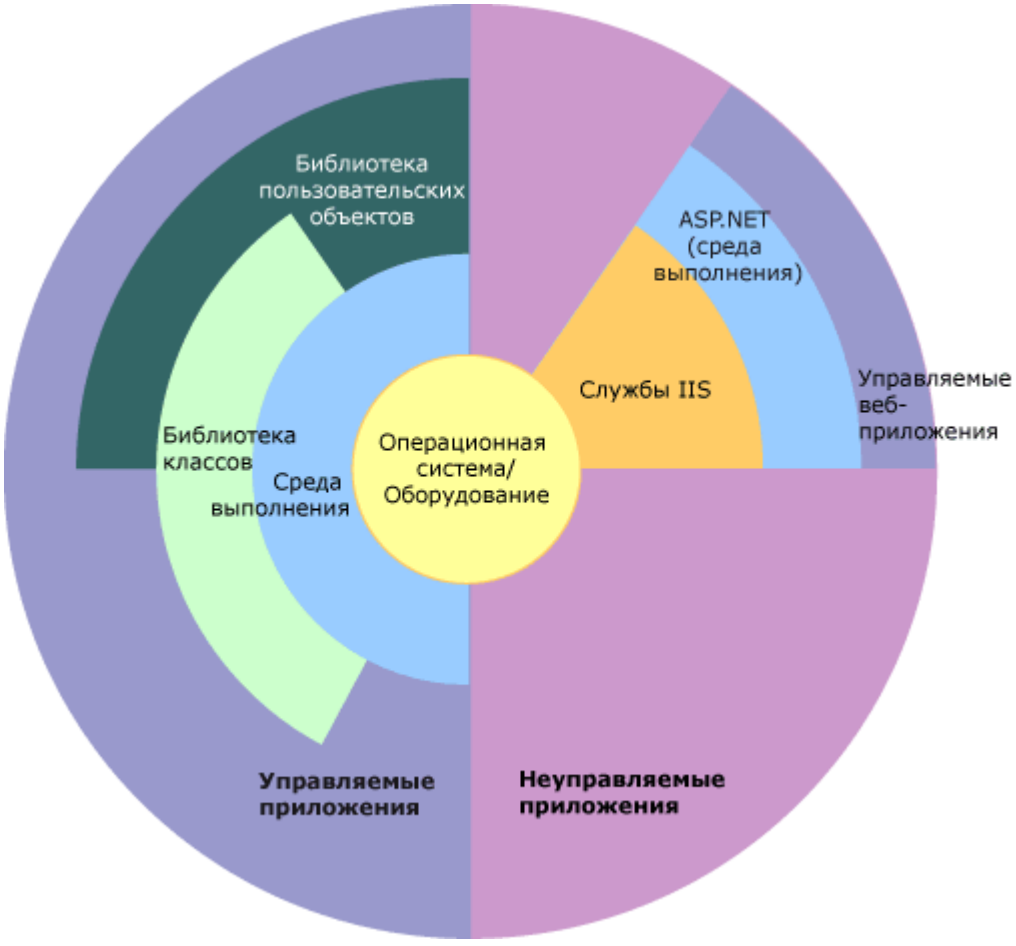
Платформа .NET базируется на двух основных компонентах: Common Language Runtime и .NET Framework Class Library.

CLR (Common Language Runtime, общеязыковая среда исполнения) – является основой, исполняющей приложения .NET, обычно написанные на CIL (Common Intermediate Language – общий промежуточный язык). Среда берет на себя работу по компиляции и выполнению кода, управлению памятью, работе с потоками, обеспечению безопасности и удалённого взаимодействия. При этом на код накладываются условия

строгой типизации и другие виды проверки точности, которые обеспечивают безопасность кода (например: если функция возвращает значение, то все «ветки» кода, которые определены в этой функции тоже должны возвращать значение; другими словами, приложение не будет компилироваться, пока не будет уверенности в том, что функция во всех ситуациях возвращает некоторое значение).

.NET Framework Class Library – универсальный набор классов, для использования в разработке программ. Она, во-первых, упрощает взаимодействие программ, написанных на разных языках, за счет стандартизации среды выполнения. Во-вторых, FCL позволяет компилятору генерировать более компактный код, что актуально при распространении программ через интернет. В терминологии .NET Framework FCL так же называют BCL (англ. Base Class Library – Базовая Библиотека Классов).

Ниже приведена обобщённая схема архитектуры платформы .NET Framework (изображение взято с сайта http://msdn.microsoft.com/, то есть эта схема, которую демонстрирует: как видят структуру Framework-а его разработчики). Схема отражает связь общеязыковой среды исполнения и библиотеки базовых классов с пользовательскими приложениями и операционной системой в целом:



## Общеязыковая среда исполнения CLR (Common Language Runtime).

Разработчикам следует понимать, что CLR является одной из реализаций спецификации, называемой общей инфраструктуры языков (Common Language Infrastructure, сокращенно CLI).

В настоящий момент ведутся работы как минимум над еще двумя реализациями, это проекты Mono (http://www.mono-project.com/) и Portable.NET (http://www.gnu.org/software/dotgnu/). Microsoft распространяет в исходных текстах еще одну свою реализацию CLI, работающую в Windows, и под управлением FreeBSD. Эта реализация называется Shared Source CLI (иногда используется ее кодовое название – Rotor).

Основными составляющими CLI являются:

- Общая система типов (Common Type System, CTS) – обеспечивает кросс-языковое взаимодействие в рамках среды .NET, охватывая большую часть типов, встречающихся в распространенных языках программирования, таких как C, C++, Visual Basic, Pascal и т.д.

- Виртуальная система исполнения (Virtual Execution System, VES) – обеспечивает загрузку и выполнение программ, написанных для CLI.

- Система метаданных (Metadata System) – служит для описания типов, используется для передачи типовой информации между различными инструментами.

- Общий промежуточный язык (Common Intermediate Language, CIL) – независимый от конкретной платформы байт-код, который выступает в роли целевого языка для любого CLI-совместимого компилятора.

- Общая спецификация языков (Common Language Specification, CLS) – набор соглашений между разработчиками языков программирования и разработчиками библиотек классов, в которых определено подмножество CTS и набор правил, направленных на обеспечение взаимодействия программ и библиотек, написанных на различных CLI-совместимых языках.

## Общая система типов CTS (common type system).

Ядро стандартной системы типов весьма обширно, так как она разрабатывалась из расчета поддержки максимального числа языков, и обеспечения максимальной эффективности работы кода на платформе .NET.

Все типы, входящие в стандартную систему типов, можно разделить на две группы: типы-значения и типы-ссылки. Главное различие между ними следующее: использование

типов-значений всегда связано с копированием их значений, а работа со ссылочными типами всегда осуществляется через их адреса.

Типы-значения существуют встроенные (к ним относятся в основном численные типы данных) и пользовательские.

Ссылочные типы описывают так называемые объектные ссылки (object references), которые представляют собой адреса объектов.

Все базовые типы данных языка C#, будут подробно рассматриваться в соответствующем разделе текущего урока.

## Общая языковая спецификация CLS (common language specification).

Общая спецификация языков (Common Language Specification, сокращенно CLS) – набор соглашений между разработчиками различных языков программирования и разработчиками библиотек классов, в которых определено подмножество CTS и набор правил. Существует общее правило межплатформенного взаимодействия в .NET:

*«Если разработчики языка реализуют хотя бы определенное в этом соглашении подмножество CTS и при этом действуют в соответствии с указанными правилами, то пользователь языка получает возможность использовать любую соответствующую спецификации CLS библиотеку. То же самое верно и для разработчиков библиотек: если их библиотеки используют только определяемое в соглашении подмножество CTS и при этом написаны в соответствии с указанными правилами, то эти библиотеки можно использовать из любого соответствующего спецификации CLS языка».*

## Библиотека классов FCL (BCL).

Библиотека FCL – один из двух «столпов» .NET Framework. FCL, по сути, является стандартной библиотекой классов платформы .NET. В литературе так же встречается название: Base Clases Library (BCL), что переводиться на русский язык как Библиотека Базовых Классов.

Программы, написанные на любом из языков, поддерживающих платформу .NET, могут использовать классы и методы FCL

К сожалению, обратное правило действует не всегда: не все языки, поддерживающие платформу .NET, обязаны предоставлять одинаково полный доступ ко

всем возможностям FCL — это зависит от особенностей реализации конкретного компилятора и языка.

FCL включает в себя следующие пространства имён (Namespaces):

- *System* – Это пространство имён включает базовые типы, такие как String, DateTime, Boolean, и другие. Обеспечивает необходимым набором инструментов для работы с консолью, математическими функциями, и базовыми классами для атрибутов, исключений и массивов.
- *System.CodeDom* – Обеспечивает возможность создавать код и запускать его.
- *System.Collections* – Определяет множество общих контейнеров или коллекций, используемых в программировании – такие как список, очередь, стек, хеш-таблица и некоторые другие. Также включена поддержка обобщений(Generics).
- *System.ComponentModel* – Обеспечивает возможность реализовывать поведение компонентов во время выполнения и во время дизайна. Содержит инфраструктуру для реализации универсальных переносимых компонентов.
- *System.Configuration* – Содержит компоненты для управления конфигурационными данными.
- *System.Data* – Это пространство имён представляет архитектуру ADO.NET, являющуюся набором программных компонентов, которые могут быть использованы для доступа к данным и для обслуживания данных.
- *System.Deployment* – Позволяет настроить способ обновления и распространения приложения с использованием технологии ClickOnce.
- *System.Diagnostics* – Предоставляет возможность диагностировать разрабатываемое приложение. Включает журнал событий, счётчики производительности, трассировку и взаимодействие с системными процессами.
- *System.DirectoryServices* – Обеспечивает лёгкий доступ к Active Directory из управляемого кода.
- *System.Drawing* – Предоставляет доступ к GDI+, включая поддержку для духмерной растровой и векторной графики, изображений, печати и работы с текстом.
- *System.Globalization* – Предоставляет помощь для написания интернационализированных приложений. Может быть определена информация, связанная с культурой, включая язык, страну/регион, календарь, шаблоны формата даты, валюты и цифр.
- *System.IO* – Позволяет осуществлять считывание и запись в различные потоки, такие как файлы и другие потоки данных. Также обеспечивает взаимодействие с файловой системой.

- *System.Management* – Предоставляет средства для запроса информации, такой как количество свободного места на диске, информации о процессоре, к какой базе данных подключено определённое приложение, и многое другое.
- *System.Media* – Позволяет проигрывать системные звуки и файлы мультимедиа.
- *System.Messaging* – Позволяет отображать и управлять очередью сообщений в сети, а также отсылать, принимать и просматривать сообщения. Другое имя для некоторых предоставленных возможностей – .Net Remoting. Это пространство имён заменено Windows Communication Foundation.
- *System.Net* – Предоставляет интерфейс для многих протоколов, используемых в сетях в настоящее время, таких как HTTP, FTP, и SMTP. Безопасность общения поддерживается протоколами наподобие SSL.
- *System.Linq* – Определяет интерфейс Iqueryable<T> и связанные с ним методы, которые позволяют подключать провайдеры LINQ.
- *System.Linq.Expressions* – Позволяет делегатам и лямбда-выражениям быть представленными как деревья выражений, так, что высокоуровневый код может быть просмотрен и обработан во время его выполнения.
- *System.Reflection* – Обеспечивает объектное представление типов, методов и свойств (полей). Предоставляет возможность динамически создавать и вызывать типы. Открывает API для доступа к возможностям рефлексивного программирования в CLR.
- *System.Resources* – Позволяет управлять различными ресурсами в приложении, используемыми, в частности, для интернационализации приложения на разных языках.
- *System.Runtime* – Позволяет управлять поведением приложения или CLR во время выполнения. Некоторые из включённых возможностей взаимодействуют с COM, сериализированными объектами внутри двоичного файла или SOAP.
- *System.Security* – Предоставляет функциональности внутренней системы безопасности CLR. Это пространство имён позволяет разрабатывать модули безопасности для приложений, базирующиеся на политиках и разрешениях. Обеспечивает доступ к средствам криптографии.
- *System.ServiceProcess* – Позволяет создавать приложения, запускаемые как сервисы в системе Windows.
- *System.Text* – Поддерживает различные кодировки, регулярные выражения, и другие полезные механизмы для работы со строками (класс StringBuilder).
- *System.Threading*


- Облегчает мультипотоковое программирование и синхронизацию потоков.
- *System.Timers* – Позволяет вызвать событие через определённый интервал времени.
- *System.Transactions* – Обеспечивает поддержку локальных и распределённых транзакций. Кроме того, в современных версиях .NET поддерживаются следующие расширения.
- Windows Presentation Foundation – для создания богатых пользовательских интерфейсов.
- Windows Communication Foundation – для простого создания сетевых приложений.
- Windows Workflow Foundation – для управления процессами выполнения.
- Windows CardSpace – для поддержки технологии «единого входа».

**Языки платформы Microsoft .NET.**

Фактически, .NET является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой Microsoft – Visual C++ .Net (с управляемыми расширениями), C#, J#, Visual Basic .Net, в среду могут добавляться любые языки программирования, компиляторы которых создаются другими компаниями-производителями. Практически компиляторы существуют для всех известных языков – Fortran и Cobol, RPG и Component Pascal, Oberon, SmallTalk и многих других.

Все разработчики компиляторов при включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение – обеспечение совместимости с CLS.

**Язык MSIL (Microsoft Intermediate Language).**

Промежуточный язык Майкрософт является целевой платформой, для которой компилируются программы в .NET Framework.

В MSIL переводят код на языках высокого уровня все компиляторы, обеспечивающие поддержку платформы .NET.

Язык MSIL напоминает ассемблер. Его можно рассматривать как ассемблер виртуальной машины .NET. В то же время язык MSIL содержит некоторые высокоуровневые конструкции, и писать код непосредственно на MSIL легче, чем на ассемблере для реальных процессоров. Поэтому его часто рассматривают как «высокоуровневый ассемблер».

Язык MSIL также называют просто IL (англ. Intermediate Language), то есть просто «промежуточный язык».

Синтаксис и мнемоника языка MSIL описываются спецификацией CIL (англ. Common Intermediate Language – Общий Промежуточный Язык) — открытой спецификацией общего промежуточного языка. Спецификация CIL является составной частью более общей спецификации CLI (англ. Common Language Infrastructure – Общая Языковая Инфраструктура).

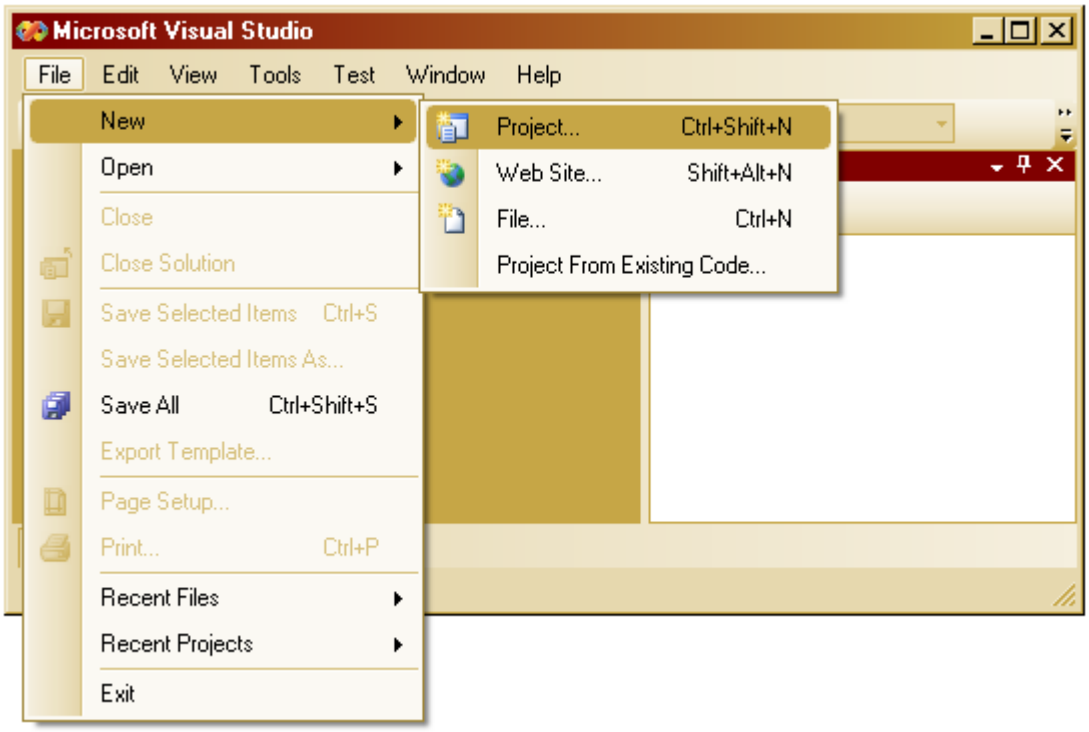
CLI – это международная спецификация, которая позиционирует идеологию языков программирования с интегрированной средой исполнения. Основной идеей является то, что приложение, имеющее составные модули, созданные на различных высокоуровневых языках программирования, переводятся не в исполняемый машинный код, а в некоторый промежуточный код, напоминающий язык ассемблера – CIL (в случае с .NET Framework – это язык MSIL). Приложение, написанное на IL – коде, компилируется в исполняемый код во время запуска, с учётом особенностей архитектуры той ЭВМ, на которой оно исполняется. Задачи компиляции возлагаются на Среду Исполнения – Language Runtime, которая, как правило, представляет собой сервис операционной системы, а сам процесс компиляции во время запуска приложения называется Just-In-Time Compilation (Компиляция «Во-Время»).

## 3. Введение в язык программирования C#.

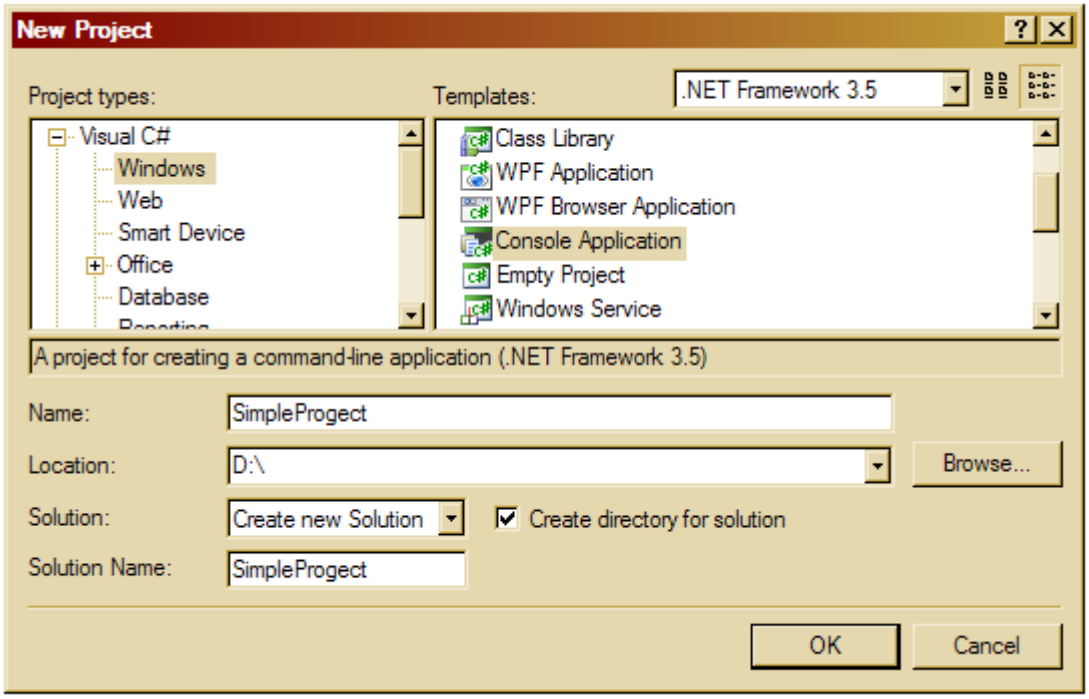
### Простейшая программа на языке программирования C#.

Приведем пример простейшей программы на языке программирования C#.

При изучении синтаксиса языка C# Вы будете работать с консольными приложениями. В качестве IDE (англ. Integrated Development Environment – Интегрированная среда разработки) Вы будете использовать Microsoft Visual Studio. Для создания консольного приложения в Visual Studio, в меню File необходимо в подменю New выбрать пункт Project (иными словами мы говорим студии, что необходимо создать новый проект).




Как реакция на нажатие пункта меню появляется диалог создания проекта.



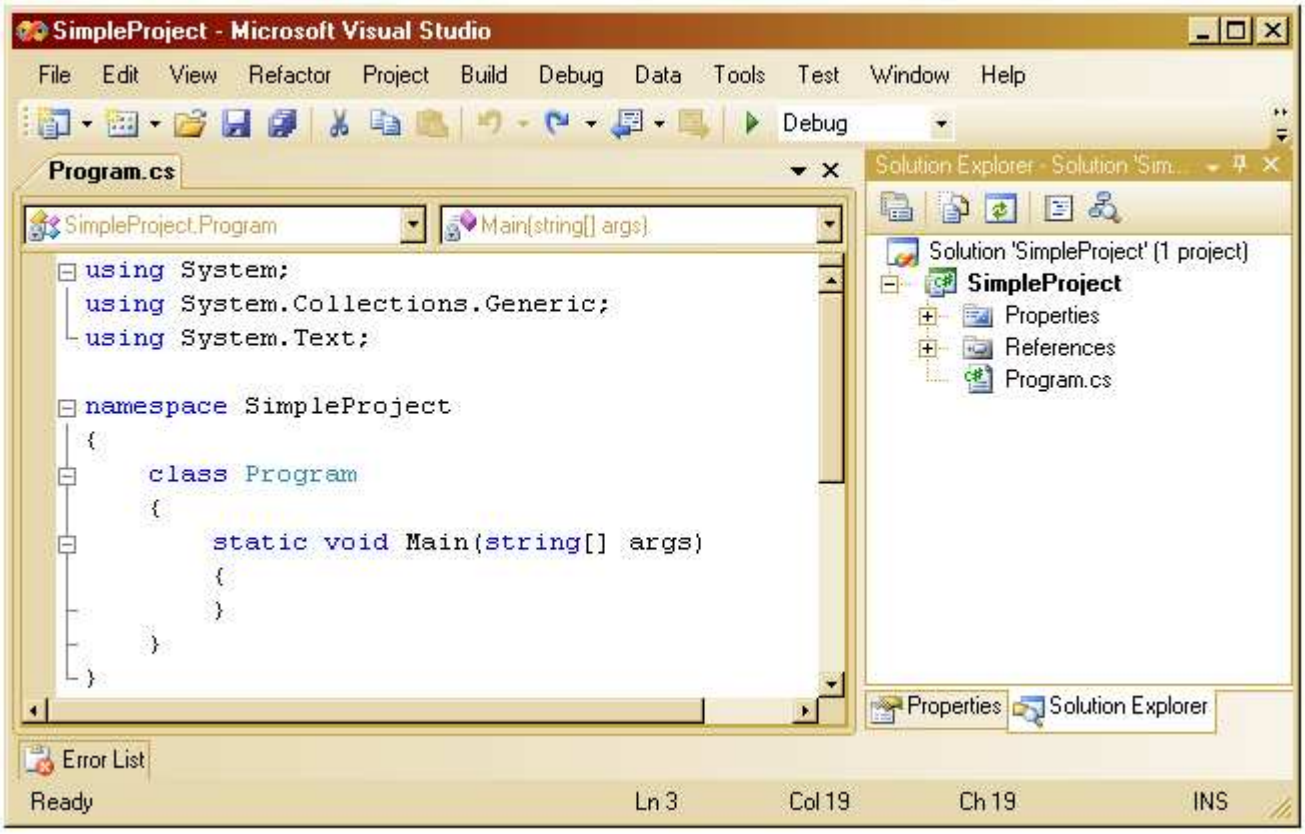

В этом диалоге в левой части в виде «дерева» представлен список доступных категорий проектов (окно **Project types** – «Типы проектов»), доступных для создания. В правой части, которая называется **Templates** (Шаблоны), представлен список шаблонов проектов, соответствующих выбранной в окне «**Project types**» категории. Нам необходимо

выбрать категорию **Visual C#->Windows**, и в этой категории выбрать шаблон **Console Application,** как показано на представленном выше рисунке.

Создание проекта по шаблону заключается в том, что среда разработки создаёт необходимый для выбранного типа проекта файлы и генерирует минимально необходимый текст. В нашем случае в проекте будет сгенерировано пространство имён название которого будет идентично названию проекта (это характерно для всех создаваемых по шаблону проектов); в этом пространстве имён будет объявлена функция Main (в дальнейшем все функции в C# мы будем называть методами, поскольку в языке C# не существует методов вне классов).

На представленной ниже картинке Вы можете видеть, что в окне справа (оно называется Solution Explorer) представлена логическая структура нашего проекта. В этом же окне вы можете получить доступ ко всем файлам и элементам, входящим в состав Вашего проекта.

Для минимального приложения необходим один файл, классически называющийся **Program.cs** (cs – это расширение файлов, хранящих исходный код на языке C#). Этот файл открывается по умолчанию в главном окне. Как показано на приведённом ниже изображении.

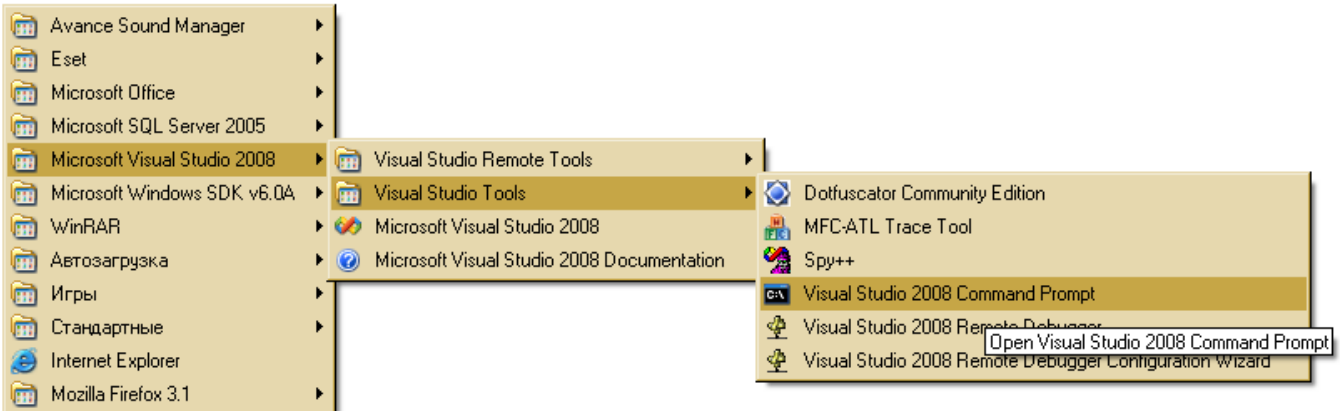



В тело функции Main необходимо добавить следующий код:

```
                Console.WriteLine(«Введите Ваше имя»);
                string name;
                name = Console.ReadLine();
                if (name == “”)
                    Console.WriteLine(“Здравствуй, мир!”);
                else
                    Console.WriteLine(“Здравствуй, “ + name + “!”);
```

Данная программа запрашивает у пользователя его имя и, если он его ввел, здоровается с ним, иначе она выводит сообщение: «Здравствуй мир!». Для ее компиляции можно воспользоваться или компилятором командной строки csc.exe или средой разработки VisualStudio. Для компиляции проекта в среде разработки Вам достаточно нажать комбинацию клавиш Ctrl+F5. После компиляции проекта сразу запуститься консоль, в которой будет выполняться откомпилированная программа.

Выполнить компиляцию так же можно при помощи компилятора командной строки csc.exe. Для этого необходимо запустить инструмент Visual Studio – Visual Studio Command Prompt. Вы можете найти её в меню «Пуск», как показано на рисунке :

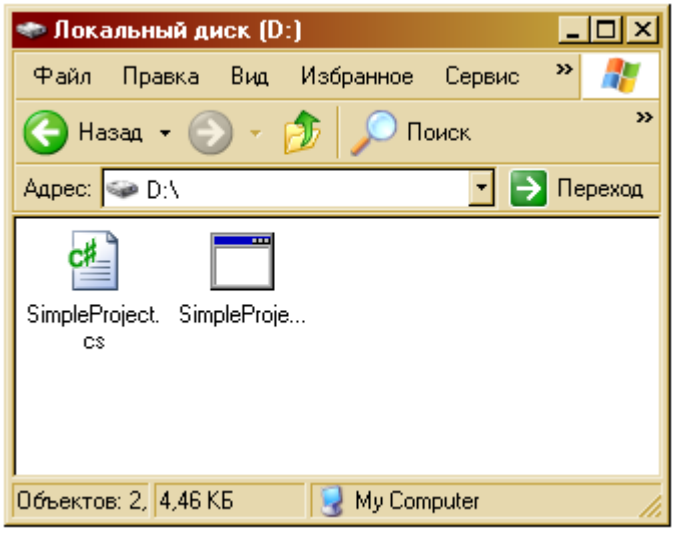



После того, как консоль запуститься, Вам необходимо выполнить следующую команду:

```
csc /t:exe /out:D:\SimpleProject.exe D:\SimpleProject.cs
```

В результате выполнения команды в корневой директории диска «D:» появиться файл «SimpleProject.exe». Хотя файл SimpleProject имеет расширение .exe – он не является обычным исполняемым файлом, содержащим машинный исполняемый код. Как уже упоминалось выше, все .NET-приложения компилируются в промежуточный код (IL). При запуске такого приложения, среда исполнения (CLR) получит сигнал, о том, что запускается .NET-приложение. Запущенный файл предаётся под управление CLR, которая, в свою очередь, инициирует отработку JIT (Just-In-Time) компилятора, переводящего файл, созданный на MSIL, в исполняемый файл (временный), написанный на машинном коде.

Именно этот временный файл и будет запущен в результате запуска файла “SimpleProject.exe”.




Далее мы рассмотрим выполненную нами команду по элементам.

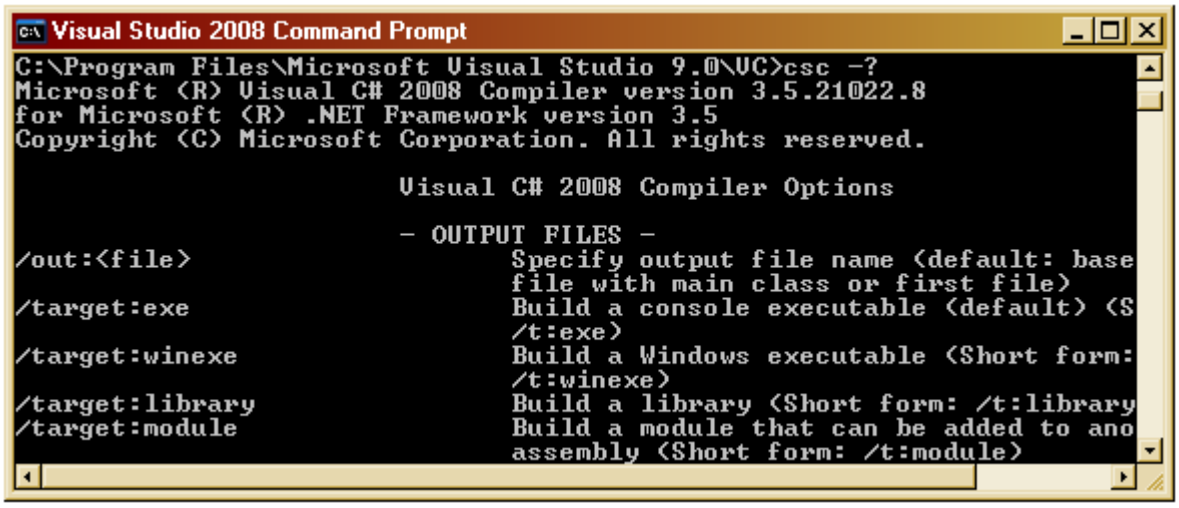
Начало команды – это название программы, которая будет выполнять компиляцию (csc). Далее следуют параметры:

- /t: (или полное название /target☺ – указывает какой тип файла должен получиться в результате компиляции (exe – консольное приложение, winexe – оконное приложение Windows, library – динамически подключаемая библиотека, module – модуль, который может быть впоследствии добавлен в другую сборку);
- /out: - указывает расположение результирующего файла;
- Последним аргументом был указан файл с исходным текстом программы.
-

Для получения подробной информации об аргументах компилятора csc необходимо выполнить команду:

```
csc -?
```

Результат этой команды будет выглядеть следующим образом:




## 4. Рефлекторы и дотфускаторы

### Что такое рефлектор?

Как упоминалось выше, проекты, созданные с использованием технологии **.NET**, компилируются не в исполняемый машинный код, а в «промежуточный код» (**IL** или **MSIL**), который легче «поддаётся» декомпиляции (восстановлению оригинального текста программы из текста исполняемого файла), в сравнении с машинным кодом. Также, технология **.NET** реализуют возможность «отражения» типов данных – получение информации о типе данных и его компонентах во время выполнения программы (в терминологии **.NET** этот процесс называется **Reflection** – «отражение»). Все вышеуказанные факты делают .NET-проекты уязвимыми с точки зрения «Права интеллектуальной собственности». Однако, на этапе разработки использование рефлектора, как программы, позволяющей просматривать содержимое .NET сборок, может быть весьма эффективным, как с точки зрения анализа программного решения, так и с точки зрения тестирования приложения.

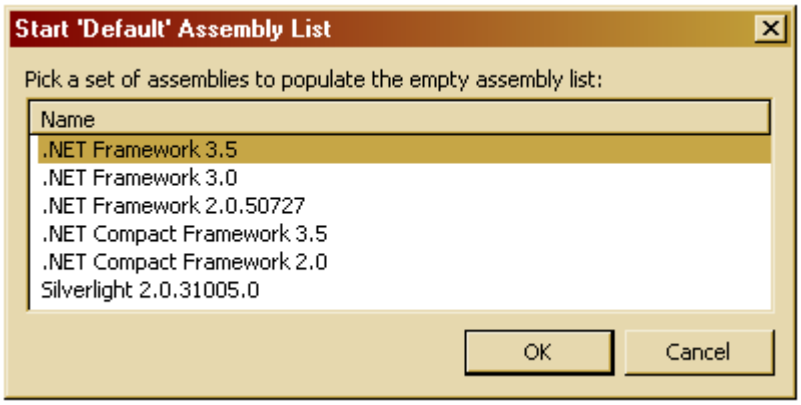
Например: при групповой разработке программист может получать от смежной группы разработчиков не исходный текст, а готовый исполняемый файл или подключаемый модуль. При необходимости просмотреть исходный текст полученного

файла можно не связываться с коллегами, для того, чтобы они передали Вам необходимый исходный файл – достаточно воспользоваться рефлектором.

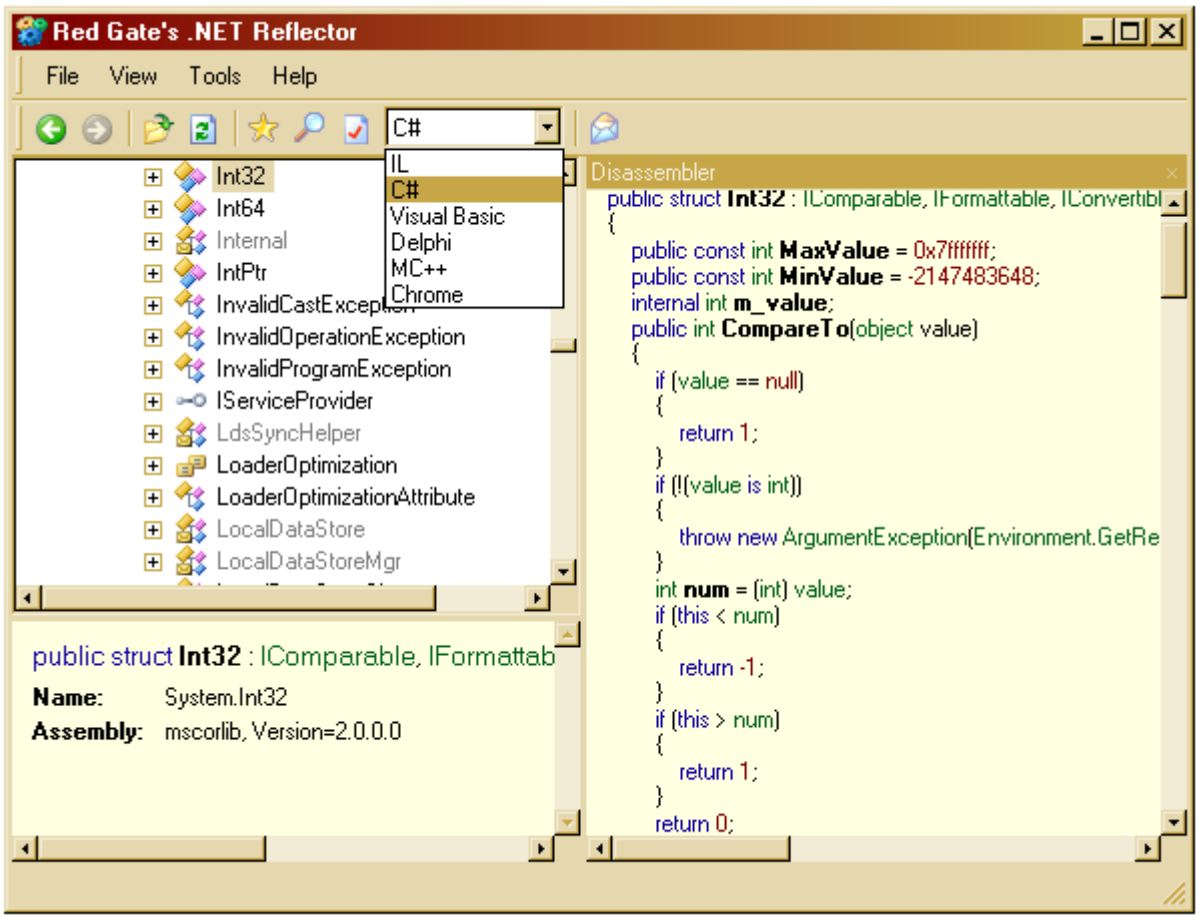
Мы рассмотрим действие рефлектора на примере приложения .NET Reflector 5.0 разработанного компанией Red Gate's. Данное приложение распространяется бесплатно и, по отзывам специалистов, является лидером среди решений в области отражения .NET сборок.

При запуске приложение предлагает Вам выбрать, какой набор сборок .NET Framework Вы хотели бы просмотреть.

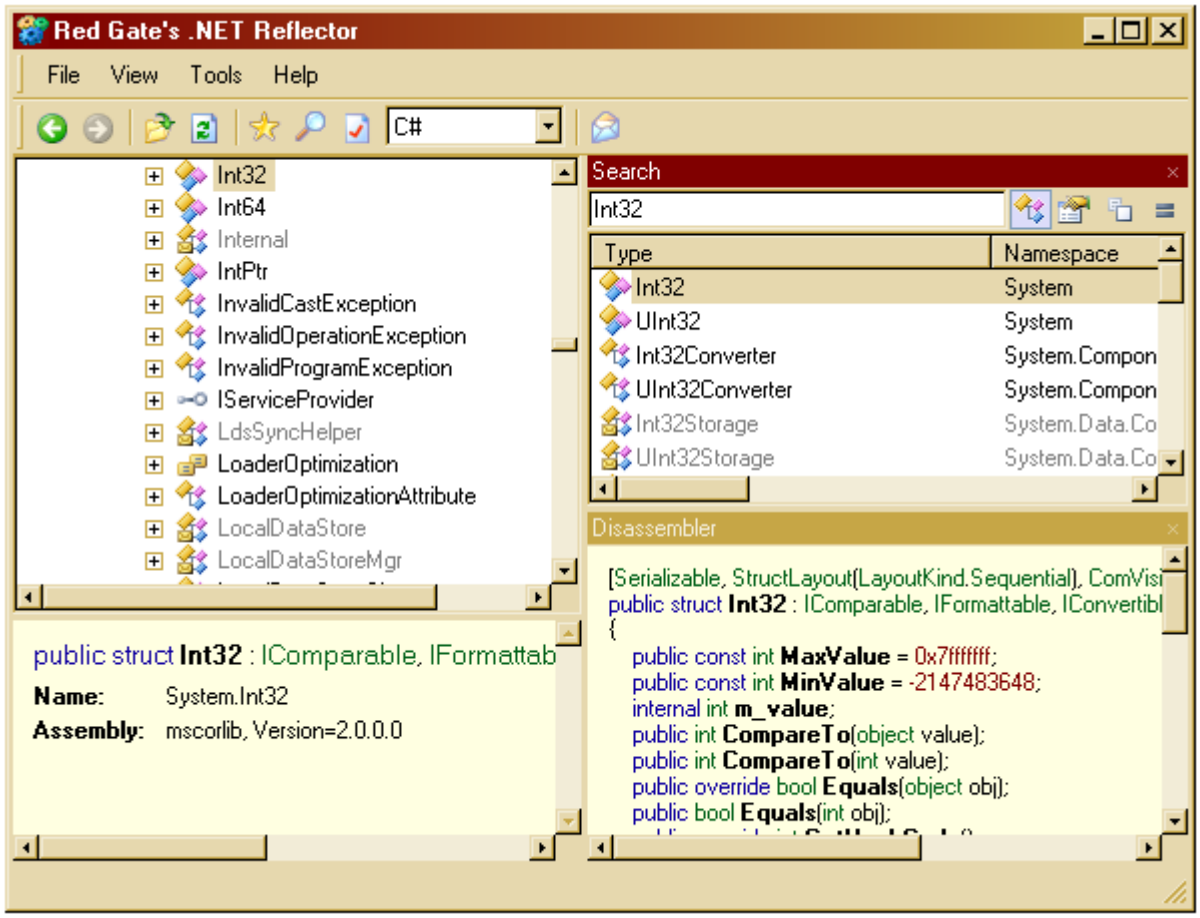
Выбранный набор сборок открывается в основном окне приложения. Левая панель основного окна содержит перечисление доступных к просмотру элементов в виде дерева. Также на этой панели (снизу) есть информационное окно, которое выводит мета-информацию о выбранном элементе (название типа данных в общей системе типов, строку объявления, а так же название и версию сборки).



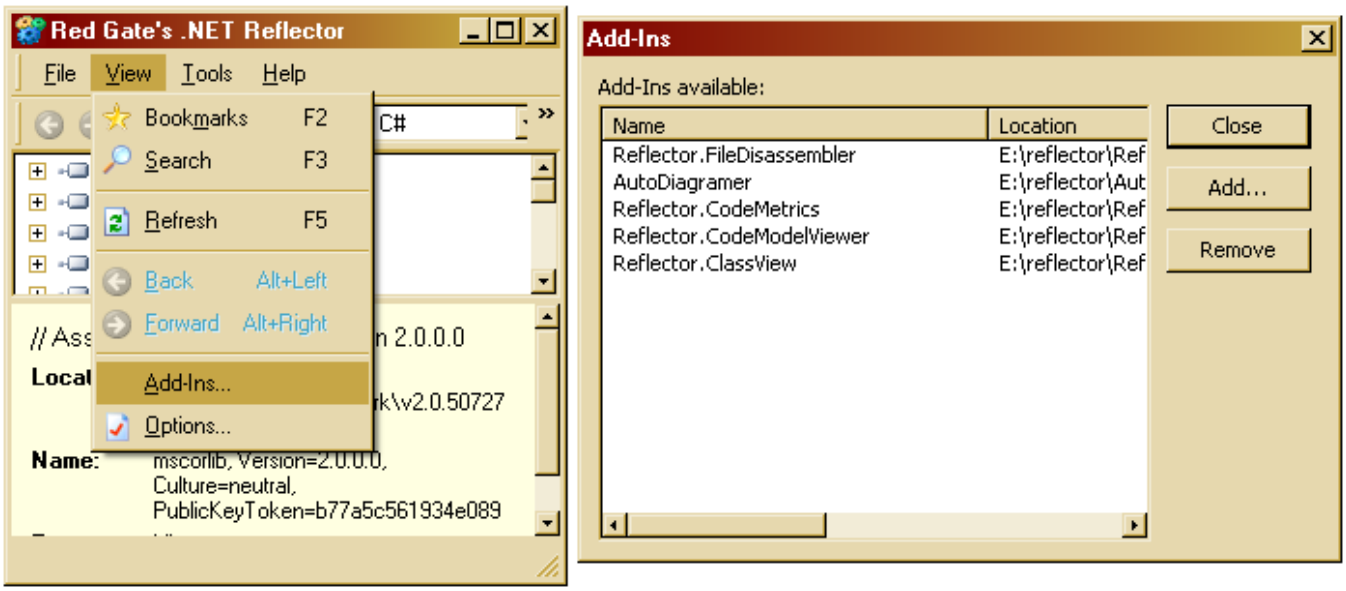

Справа в главном окне доступно окно **Disassembler**, в котором открывается полное описание выбранного в левой панели класса, включая объявление, компоненты и описание методов. Содержимое окна Disassembler будет представлено на том языке программирования, который выбран Вами из выпадающего списка, показанного на приведённом ниже изображении.



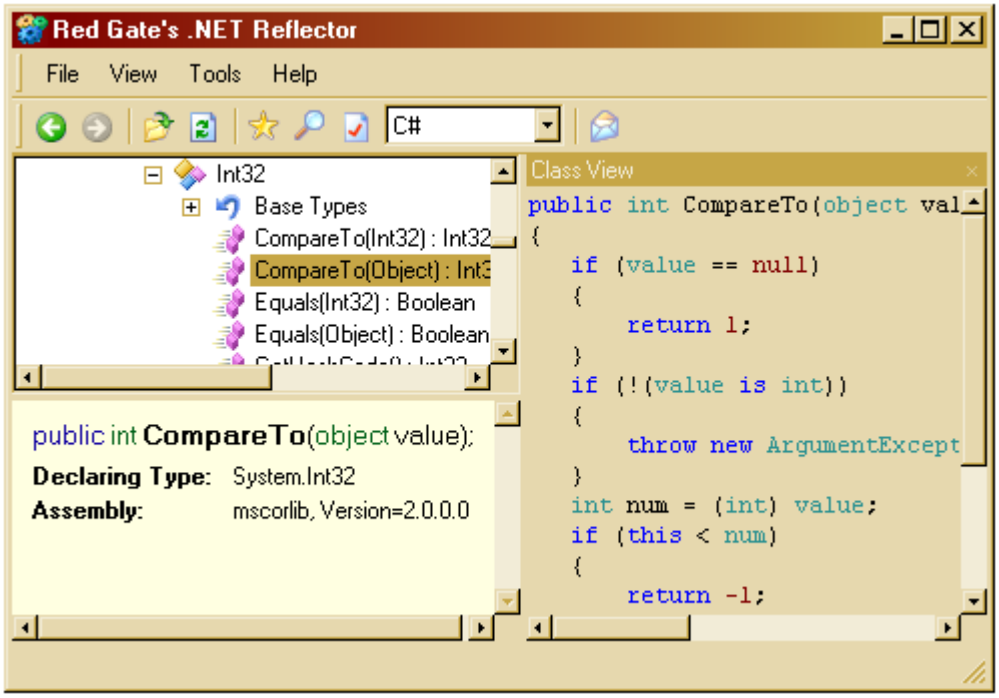

На следующем изображении показано окно **Search** (поиск), которое позволяет получить список элементов по их идентификатору (имени).




.NET Reflector так же поддерживает дополнения (add-ins), которые можно бесплатно скачать на официальном сайте проекта (http://reflector.red-gate.com/). Добавлять дополнения в рефлектор нужно следующим образом: файл (или несколько файлов) дополнения, который вы скачали, необходимо поместить в директорию, в которой находиться исполняемый файл .NET Reflector-а. Далее, необходимо выбрать пункт Add-Ins в меню View, после чего появиться диалог управления дополнениями. Управление дополнениями происходит посредством нажатия соответствующих кнопок (доступны действия: добавить и убрать). После добавления дополнений в меню Tools, обычно (возможны и другие варианты), интегрируется специальный пункт для каждого дополнения. Диалог управления дополнениями показан на приведённом ниже рисунке:




На приведённом ниже рисунке мы открыли окно дополнения, которое называется **ClassView.** Его назначение состоит в представлении текста кода в том виде, в котором он представлен в Visual Studio, что является более привычным и увеличивает читабельность текста.



## Необходимость использования рефлектора.

Использование рефлектора предоставляет возможность увидеть, как выглядит сборка после её компиляции. Так же, рефлектор позволяет просмотреть код, который был сгенерирован компилятором, что является важным при использовании различных надстроек и расширений языка (например LINQ), которые позволяют увеличить гибкость языка, но в то же время разработчик не всегда знает (особенно при отсутствии опыта), какой код получится в итоге. Так же (как уже упоминалось выше), рефлектор может использоваться как средство анализа при тестировании продукта. Так же при наличии специальных дополнений рефлектор может позволять осуществлять анализ содержимого сборки на наличие дублирующих элементов кода и многое другое.

## Что такое дотфускатор?

Названные в предыдущем разделе особенности технологии .NET Framework, в частности возможность отражения кода, таит в себе, как уже упоминалось, сложности со стороны защиты интеллектуальных прав разработчика (владельца интеллектуальных прав на программный продукт). Программные продукты, которые легко декомпилируются, открывают все возможности **обратной инженерии** (процессу воссоздания изделия (продукта) по его экземпляру).

Однако и открытый код можно защитить. Для этой цели были созданы программные продукты под общим названием дотфускаторы (**dotfuscators**), название **дотфускатор** происходит из названия основного подхода к защите программных решений– **обфускации** (**obfuscation**) или в переводе на русский язык – **запутывания**. Данный процесс состоит из набора методов запутывания кода. Каждый метод фокусируется на определённом виде информации. Таким образом выделяют запутывание разметки кода, запутывание данных, запутывание конструкций управления

кодом и превентивную трансформацию. Методы запутывания необходимо использовать комплексно. Если для защиты используется только один метод, защиту достаточно просто обойти. А цель процесса запутывания состоит в том, чтобы, при сохранённой функциональности, сделать сборку нечитаемой.

Запутывание разметки состоит в изменении форматирования исходного кода. Частным случаем изменения разметки является переименование. Переименование – самый простой метод запутывания, состоящий в замене используемых в приложении мнемонических идентификаторов на немнемонические. Таким образом, тому, кто декомпилирует сборку, будет достаточно сложно понять, что происходит в рамках программы.

Запутывание данных фокусируется на структурах данных, которые использует приложение. Выделяют следующие методы запутывания данных:

- **запутывание размещения данных в памяти** (например конвертирование локальных переменных в глобальные);
- **запутывание агрегации данных**, состоящее в запутывании способов группировки данных;
- **запутывание порядка**, состоящее в изменении порядка элементов в соответствии с определённым правилом.

Запутывание конструкций управления фокусируется на конструкциях управления кодом. Выделяют следующие методы запутывания конструкций управления:

- **запутывание агрегации**, состоящее в запутывании взаимного расположения выражений в программе;
- **запутывание порядка**, состоящее в изменении порядка исполнения выражений;
- **запутывание вычислений**, состоящее в запутывании вычислений производимых в приложении (например добавление в приложение элементов, которые никогда не будут выполняться, что усложнит понимание кода).

Методы превентивной трансформации состоят в том, чтобы усложнить процесс обратный запутыванию (**распутывание** или **деобфускация**). Эти методы совершенствуют методы запутывания, основываясь на достижениях в области деобфускации.

## Необходимость использования дотфускаторов.

Необходимость использования дотфускаторов очевидна. Код программных решений необходимо защищать от методов обратной инженерии, особенно если эти решения являются коммерческими. Поскольку в современных условиях невозможно полностью контролировать распространение экземпляров продукта, выпущенного на рынок, и разработчик никогда не знает кому попадёт его решение.

Запутывание же усложняет процесс обратной-инженерии, с одной стороны, чем защищает интеллектуальные права разработчика и его коммерческую тайну. С другой стороны, процесс запутывания усложняет работу по анализу средств безопасности приложения, что уменьшает вероятность взлома программного решения пиратам.

## Обзор существующих дотфускаторов.

Ниже приведено перечисление некоторых из существующих на сегодняшний день дотфускаторов:

1. Dotfuscator Community Edition – дотфускатор, который входит в базовую поставку Visual Studio, однако этот дотфускатор «умеет» только выполнять «переименование», что нельзя назвать надёжной защитой, например, если основным ноухау является некоторый алгоритм приложения. Чтобы в Dotfuscator Community Edition включились основные функции его нужно обновить до версии Dotfuscator Professional Edition, который распространяется на коммерческих основаниях и стоит недёшево.

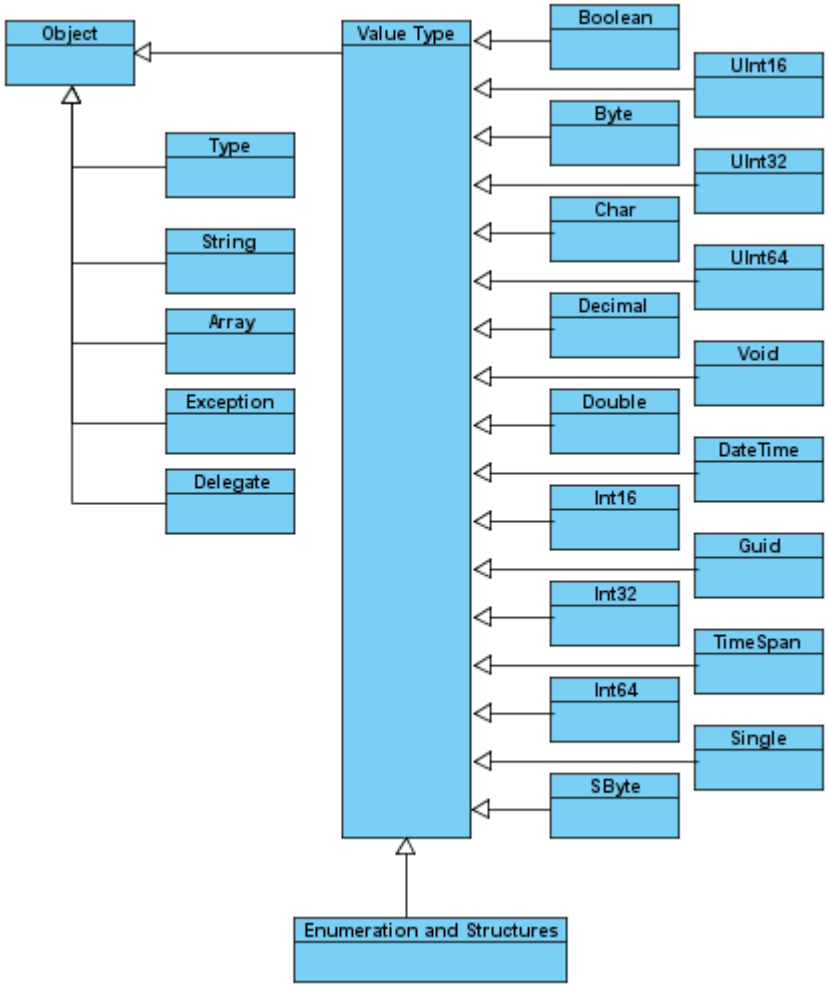
2. AssemblyLockbox - сервис компании Gibwo. Стоимость этого решения составляет $49.95 в месяц. Компания позиционирует альтернативное решение защиты кода – защита через шифрование. Они предоставляют свой сервис, который кодирует сборку и декодирует её обратно в момент выполнения. Однако их сервис имеет один недостаток – необходимость наличия их сервиса программного модуля на компьютере клиента, без которого дешифровка производиться не будет.
3. Babel - программный продукт компании Alberto Ferrazzoli, который распространяется по лицензии GNU Lesser General Public License (то есть бесплатно). Представляет собой консольное приложение, реализованное на базе Microsoft Phoenix framework (framework, который предназначен для создания компиляторов и различных инструментов анализа приложений, их оптимизации и тестирования). Данное решение поддерживает Microsoft .NET Framework до 3.5, а так же все основные методы запутывания кода.
4. C# Source Code Obfuscator – коммерческое решение компании Semantic Designs стоимостью в $200. Поддерживает все основные методы запутывания и может иметь как консольный, так и графический интерфейс. Поставляется так же в виде комплексного решения, содержащего средства анализа программных продуктов, средства тестирования, и, конечно, обфускации. Стоимость такого продукта составляет 1000$.
5. CodeVeil – дотфускатор компании Xheo. Как и компания Gibwo, компания Xheo позиционирует идею шифрования, для обеспечения безопасности кода. Идея состоит в том, что в исполняемый файл добавляется специальный код, который дешифрует приложение перед выполнением. Стоимость Professional версии составляет 1199$.

## 3. Типы данных

Тип данных определяют множество значений, которые может принимать объект (экземпляр этого типа), множество операций, которые допустимо выполнять над ним, а так же способ хранения объектов в оперативной памяти.

Технология **.NET Framework** определяет две группы типов данных: значащие типы или типы значений (value-types) и ссылочные типы (reference-types). Определено, что экземпляры значащих типов должны располагаться в стеке, тогда как ссылочных – в управляемой оперативной памяти (куче), в терминологии **.NET Framework** называемой managed heap.

*Общая иерархическая структура Базовой Системы Типов*




В **.NET Framework** все, даже самые простые, типы данных представлены некоторым классом или структурой в общей иерархической структуре классов. Необходимо помнить, что тип, находящийся на вершине иерархической структуры, определяет поведение производных от него классов. Иерархическая структура в общем виде представлена на приведённом выше рисунке.

Для описания значащих типов данных используются структуры. Тогда как для описания ссылочных – классы. Таким образом любому базовому типу данных соответствует объявление некоторой структуры (например типу int соответствует структура System.Int32).

### Целочисленные типы данных

В приведённой ниже таблице можно видеть, что в C# определены обе версии всех целочисленных типов данных: со знаком и без знака. Так же в таблице для каждого базового типа указан соответствующий этому типу **.NET** класс, размер в битах и диапазон возможных значений.

| Название | .NET Class | Наличие знака | Размер в битах | Диапазон значений |
|---|---|---|---|---|
| **byte** | Byte | - | 8 | от 0 до 255 |
| **sbyte** | Sbyte | + | 8 | от -128 до 127 |
| **short** | Int16 | + | 16 | от -32 768 до 32 767 |
| **ushort** | Uint16 | - | 16 | от 0 до 65 535 |
| **int** | Int32 | + | 32 | от -2 147 483 648 до 2 147 438 648 |
| **uint** | Uint32 | - | 32 | от 0 до 4 294 967 295 |
| **long** | Int64 | + | 64 | от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 |
| **ulong** | Uint64 | - | 64 | от 0 до 18 446 744 073 709 551 615 |

### Типы данных для чисел с плавающей точкой

Для операций над числами с плавающей точкой определены три типа данных, которые указаны в следующей таблице

| Название | .NET Class | Наличие знака | Размер в битах | Диапазон значений |
|---|---|---|---|---|
| **float** | Single | + | 32 | от 3.402823e38 до 3.402823e38 |
| **double** | Double | + | 64 | от -1.79769313486232e308 до 1.79769313486232e308 |
| **decimal** | Decimal | + | 128 | от ±1.0 × 10e−28 до ±7.9 × 10e28 |

Особого внимания заслуживает тип данных decimal, который предусмотрен для вычислений, требующих особой точности в представлении дробной части числа, а так же минимизации ошибок округления. Тип данных decimal, в основном, используется для финансовых вычислений.

Decimal не устраняет ошибки округления, но минимизирует их. При делении единицы на 3 мы получаем число в периоде (иными словами теряем часть информации). После умножения результата на 3 мы не получим единицу, однако мы получаем число, максимально близкое к единице (опять же периодическое). Этот процесс, в сравнении с аналогичным процессом но с использованием типа double, демонстрирует следующий пример:

*Примечание:*

*В этом и последующих примерах мы будем использовать методы вывода данных в консоль:* ***Write*** *и* ***WriteLine****, которые инкапсулированы в классе Console. Эти методы являются статическими, поэтому для их вызова не нужно создавать экземпляр класса Console. Метод* ***Write*** *осуществляет вывод данных в консоль без перевода каретки. Тогда как метод* ***WriteLine*** *отличается тем, что добавляет к выведенным данным символ перехода на следующую строку (символ «\n»).*

```
Decimal devidend = decimal.One;
//нижеследующая строка выводит в консоль 1
Console.WriteLine(devidend);
decimal devisor = 3;
devidend = devidend / devisor;
//нижеследующая строка выводит в консоль //0,33333333333333333333333333
Console.WriteLine(devidend);
//нижеследующая строка выводит в консоль //0,99999999999999999999999999
//из чего можно сделать вывод, что ошибки округления //привели к потере данных
Console.WriteLine(devidend * devisor);
double doubleDevidend = 1;
//нижеследующая строка выводит в консоль 1
Console.WriteLine(doubleDevidend);
System.Double doubleDevisor = 3;/*в данной строке объявлена переменная типа
double. Использование выражения System.Double идентично использованию ключевого
слова double. Отличие состоит в том, что мы явно указываем структуру (тип
данных).*/
doubleDevidend = doubleDevidend / doubleDevisor;
//нижеследующая строка выводит в консоль //0,33333333333333
Console.WriteLine(doubleDevidend);
//нижеследующая строка выводит в консоль 1
Console.WriteLine(doubleDevidend * doubleDevisor); //при использовании типа
double мы получаем потерю //информации в обоих направлениях
```

Однако может возникнуть необходимость округления переменной типа decimal. В таком случае необходимо использовать метод Round класса Math, как показано на нижеследующем примере.

```
Decimal devidend = decimal.One;
decimal devisor = 3;
 //нижеследующая строка выводит в консоль 1
Console.WriteLine(Math.Round(devidend / devisor * devisor));
```

Класс Math инкапсулирует в себе наиболее используемые и необходимые методы и константы, связанные с математическими вычислениями. Такие методы как тригонометрические функции, возведение в степень, вычисление квадратного корня, получение абсолютного значения числа, константу Пи и так далее. Так как все эти компоненты в классе Math являются статическими, то нет необходимости создавать экземпляр этого класса, а достаточно только обратиться к его компоненту, как это показано на предыдущем примере.

## Символьный тип данных

В .NET Framework тип данных char используется для выражения символьной информации и представляет символ в формате Unicode. Формат Unicode предусматривает, что каждый символ идентифицируется 21-битным скалярным значением, называемым “code point” («кодовая точка» или «кодовый пункт»), и предусматривает кодовую форму UTF-16, которая определяет, как кодовая точка декодируется в последовательность из одного или более 16-битного значения. Каждое 16-битное значение находиться в диапазоне от шестнадцатиричного 0x0000 до 0xFFFF и располагается в структуре Char.

Таким образом, значение объекта типа char – это его 16-битное числовое положительное значение.

Структура Char предоставляет методы для сравнения объектов типа char, конвертирования текущего char-объекта в объект другого типа, преобразования регистра символа, а так же определения категории текущего символа. Некоторые методы демонстрируются на следующем примере.

```
/*Описание действия метода:                          Результат:
-------------------------------------------------*/
//определяет является ли символ управляющим
Console.WriteLine(char.IsControl('\t'));          //true
//определяет является ли символ цифрой
Console.WriteLine(char.IsDigit('5'));             //true
//определяет является ли символ буквенным
Console.WriteLine(char.IsLetter('x'));            //true
//определяет находится ли символ в нижнем регистре
Console.WriteLine(char.IsLower('m'));             //true
//определяет находится ли символ в верхнем регистре
Console.WriteLine(char.IsUpper('P'));             //true
//определяет является ли символ числом
Console.WriteLine(char.IsNumber('2'));      //true
//определяет является ли символ разделителем
Console.WriteLine(char.IsSeparator('.'));         //false
//определяет является ли символ специальным символом
Console.WriteLine(char.IsSymbol('<'));            //true
//определяет является ли символ пробелом
Console.WriteLine(char.IsWhiteSpace(' '));  //true
```

```
//переводит символ в нижний регистр
Console.WriteLine(char.ToLower('T'));            //t
//переводит символ в верхний регистр
Console.WriteLine(char.ToUpper('t'));            //T
```

## Другие типы данных

Среди перечисленных выше базовых типов данных в .NET Framework предусмотрены такие типы как object и string .

Тип данных string (.NET Class: System.String) представляет собой коллекцию Unicode символов. Коллекции – это, с одной стороны, синтаксическая категория языка C#, а с другой способ представления множеств (массивов) в рамках языка. Данная глава не ставит целью рассмотрение коллекций, речь о которых пойдёт в последующих уроках. Достаточно будет, пока, представлять string как массив символов в формате Unicode.

Класс String содержит совокупность следующих статических методов:

- **Compare** – сравнивает указанные подстроки двух переданных в качестве аргументов строк
- **CompareOrdinal** – действует аналогично предыдущему методу, но за тем исключением, что если для метода Compare символ 'a' является меньшим, чем символ 'A', то для CompareOrdinal 'a' больше, чем 'A'.
- **Concat** – возвращает результат объединения переданного массива строк.
- **Copy** – создаёт новый экземпляр строки.
- **Equals** – сравнивает значение двух строк.
- **Format** – используется для замещения помещённых в строку маркеров строковыми эквивалентами одного или более значений (по действию аналогичен методу Format класса String библиотеки MFC).

- **IsNullOrEmpty** – определяет является ли значение строковой переменной пустым или эквивалентно значению null.

А так же нестатических методов:

- **Clone** – возвращает ссылку на новый экземпляр строки равной текущей строке.
- **Contains** – определяет содержит ли текущая строка указанное строковое значение.
- **CopyTo** – копирует текущую строку в указанный массив символов.
- **EndsWith** – определяет заканчивается ли текущая строка указанным строковым значением.
- **Insert** – вставляет подстроку в указанную позицию в текущей стороке.
- **Remove** – удаляет из текущей строки все вхождения указанной подстроки.
- **Replace** – заменяет указанную подстроку новой подстрокой.
- **Split** – делит строку на массив строк, основываясь на переданном массиве разделителей.
- **StartsWith** – определяет начинается ли текущая строка указанной подстрокой.
- **EndsWith** – определяет заканчивается ли текущая строка указанной подстрокой.
- **Substring** – возвращает указанную подстроку текущей строки.
- **ToLower** – переводит текущую строку в нижний регистр.
- **ToUpper** – переводит текущую строку в верхний регистр.
- **Trim** – удаляет пробельные символы с обоих концов строки.
- **TrimEnd** – удаляет пробельные символы с конца строки.
- **TrimStart** – удаляет пробельные символы сначала строки.

Использование некоторых методов демонстрируется на следующем примере:

```
string str = “hello ☺”;
object anotherString = str.Clone();
Console.WriteLine(anotherString);
//выводит: “hello ☺”
Console.WriteLine(str.Contains(“hello”));
//выводит: true
```

```
Console.WriteLine(str.Insert(6, “world”));
//выводит: “hello world☺”
Console.WriteLine(str.Remove(5, 1));
//выводит: “hello☺”
Console.WriteLine(str.Replace(“☺”, “☹“));
//выводит: “hello ☹“
Console.WriteLine(str.StartsWith(“hell”));
//выводит: “true”
Console.WriteLine(str.Substring(6));
//выводит: “☺”
Console.WriteLine(str.ToUpper());
//выводит: “HELLO ☺”
str = “       ” + str + “       “;
Console.WriteLine(str.TrimEnd());
//выводит: “       hello ☺”
str = “       ” + str + “       “;
str = str.Trim();
Console.WriteLine(str);
//выводит: “hello ☺”
Console.WriteLine(string.Format(“the message is \”{0}\” and some numeric
value is {1}”, str, 512));
/*выводит: “the message is “hello ☺” and some
numeric value is 512”*/
```

Обратите внимание на то, что методы не изменяют текущую строку, а возвращают новый экземпляр изменённой строки. Поэтому для сохранения изменений необходимо присваивать результат работы метода исходной строке как показано ниже:

```
string str = «hello ☺»;
str = “       ” + str + “       “;
str = str.Trim();
```

В свою очередь, тип данных object (.NET Class: System.Object) так или иначе является базовым для всех ссылочных типов данных и имеет пять методов, которые наследуются производными от него классами:

- **ToString** – возвращает значение экземпляра в строковом виде, либо название типа в строковом виде, если получить строковую интерпретацию значения невозможно;
- **GetType** – возвращает тип текущего объекта;
- **Equals** – возвращает True, если два объекта, переданные в качестве аргументов, тождественны;

- **ReferenceEquals** – возвращает True, если, переданная в качестве аргумента ссылка на объект, ссылается на текущий объект;
- **GetHashCode** – служит в качестве хеш-функции текущего типа.

Очень удобным является наследование метода ToString(), что делает все типы приводимыми к их строковому эквиваленту. Однако, для корректности желательно переопределять поведение этого метода в производных классах, поскольку для ссылочных типов этот метод по умолчанию возвращает имя типа данных в системе типов .NET Framework. Следующий пример демонстрирует этот факт:

```
using System;
namespace ToStringSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Program prg = new Program();
            Console.WriteLine(prg.ToString());
            /*В консоль выводиться текст:
             ToStringSample.Program
            что соответствует имени
             основного класса нашей программы
            в системе типов .NET Framework*/
        }
    }
}
```

## 6 Литералы

Литералы в C# - это фиксированные значения, которые представлены в понятной форме. Следуя традиции языков семейства «C» литералы так же можно называть константными значениями. Так, например, 100 – это целочисленный литерал (константное целое число).

Так как C# - строго типизированный язык, то все литералы должны иметь тип. Может возникнуть вопрос: «А каким образом компилятор определяет к какому типу

отнести тот или иной литерал?». На этот случай в языке определено несколько правил идентификации типов литералов.

Для целочисленного литерала будет назначен наименьший целочисленный тип, который позволит его хранить. Все дробные значения будут иметь тип double. Необходимо отметить, что для того чтобы число считалось дробным оно обязательно должно иметь целую часть, дробную часть и десятичную точку. Например таким образом «1.5». Если понадобиться определить дробное число без дробной части. Другими словами, необходимо объявить литерал 5 таким образом, чтобы он был определён как double. То необходимо указать дробную часть как «0», например так, «5.0».

Для явной спецификации типа данных литерала в C# предусмотрены специальные суффиксы. Таким образом литера:

- объявленный с суффиксом **«L»** будет иметь тип long;
- с суффиксом **«F»** будет иметь тип float;
- с суффиксом **«D»** будет иметь тип double;
- с суффиксом **«M»** будет иметь тип decimal;
- суффикс **«U»** делает число беззнаковым (суффикс **«U»** может быть объединён с суффиксами, специфицирующими тип данных).

Использование суффиксов демонстрируется на следующем примере:

```
/*при помощи метода GetType()
программа возвращает тип данных литералов,
демонстрируя действие суффиксов
----------------------------------------*/
Console.WriteLine((10D).GetType());
/*выводит в консоль: System.Double
что соответствует типу данных double*/
Console.WriteLine((10F).GetType());
/*выводит в консоль: System.Single
что соответствует типу данных float*/
Console.WriteLine((10M).GetType());
/*выводит в консоль: System.Decimal
что соответствует типу данных decimal*/
Console.WriteLine((10).GetType());
/*выводит в консоль: System.Int32
что соответствует типу данных int*/
Console.WriteLine((10L).GetType());
/*выводит в консоль: System.Int64
```

```
        что соответствует типу данных long*/
        Console.WriteLine((10UL).GetType());
        /*выводит в консоль: System.Uint64
        что соответствует типу данных ulong*/
        Console.WriteLine(0xFF);
        /*выводит в консоль: 255
        шестнадцатиричное число 0xFF соответствует
        десятичному числу 255*/
```

В отдельную группу литералов выделены управляющие символьные последовательности, которые не имеют символьного эквивалента, а преимущественно используются для форматирования текста:

| символ | Действие управляющего символа |
|---|---|
| \a | Звуковой сигнал |
| \b | Возврат на одну позицию |
| \f | Переход к началу следующей страницы |
| \n | Новая строка |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \0 | Нуль-символ (символ конца строки) |
| \’ | Одинарная кавычка |
| \” | Двойная кавычка |
| \\ | Обратная косая черта |

Строковые литералы или константные строки выражаются в виде текста, заключённого в двойные кавычки. Например “Hello world☺” – это строковый литерал. Для форматирования теста используются в основном управляющие символы. Однако возможно указать режим «буквального» (verbatim) форматирования, при котором можно не переходить на новую строку без использования управляющих символов. В таких строковых литералах всё содержимое интерпретируется как символы (в том числе и управляющие символы). Для этого перед строковым литералом необходимо указать символ **@** (строка @”hello world” является буквально отформатированной строкой). Использование строковых литералов демонстрируется на следующем примере:

```
Console.WriteLine(«Некоторое простое сообщение\nИ»+ «ещё одно простое
сообщение на новой строке»);
/*выводит в консоль следующее сообщение:
«Некоторое простое сообщение
И ещё одно простое сообщение на новой строке»*/
Console.WriteLine(«Пример табуляции: «+ «\n1\t2\t3\n4\t5\t6»);
/*выводит в консоль следующее сообщение:
«Пример табуляции:
1  2   3
4  5   6»*/
Console.WriteLine(@»Пример буквального строкового литерала:
1       \t     3
\n   5       6»);
/*выводит в консоль следующее сообщение:
«Пример буквального строкового литерала:
       1      \t     3
          \n   5       6»*/
```

# 7 Переменные

## Понятие переменной

Переменная – это именованный объект, хранящий значение некоторого типа данных. Язык C# относиться к «**type-safe**» языкам. Иными словами, компилятор C# (**C# compiler**) гарантирует нам, что расположенное в переменной значение всегда будет одного и того же типа. Так как C# является **строго типизированным** (**strongly typed**)

языком программирования, то при объявлении переменной обязательно нужно указывать её тип данных. В общем виде объявление переменной выглядит следующим образом:

```
Тип_данных имя_переменной;
Тип_данных имя_переменной = инициализирующее значение;
```

Важным моментом является **начальное значение** (**initial value**) переменной. Дело в том, что согласно синтаксису языка C# переменная должна быть обязательно инициализирована перед использованием.

*Примечание:*

*В спецификации языка C# в главе 5 («Variables») указано следующее: « A variable must be definitely assigned (Section 5.3) before its value can be obtained». Дословный перевод звучит как: «Переменной должно быть явно присвоено значение до того, как её значение может быть получено». Данный тезис говорит нам о том, что первым действием над переменной в рамках программы может быть только присвоение ей значения, которое и называется стартовым (initial value).*

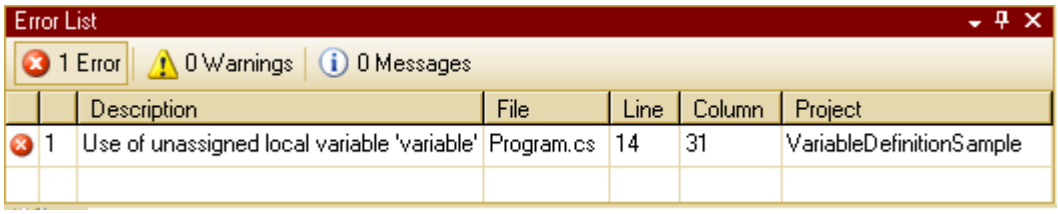
Объявление целочисленной переменной и её последующая инициализация выглядит следующим образом:

```
int variable;
variable = 5;
Console.WriteLine(variable);
//выводит в консоль: 5
```

А попытка получения значения из не инициализированной переменной, как на примере:

```
int variable;
Console.WriteLine(variable);
```

возвращает ошибку **компиляции:** «Use of unassigned local variable» («Использование не инициализированной локальной переменной»). Ошибки, возникающие на этапе компиляции, среда разработки представляет в окне «Error list», как показано на рисунке ниже:




**Правила именования переменных**

Правила именования переменных обычно отождествляют с вопросом нотации языка.

> *Примечание:*
>
> *Нотация понимается, как множество допустимых символов и правила их применения, используемые для представления лексических единиц и их взаимоотношений.*

Говоря о правила именования подразумевают не только имена переменны, но все «имена» (названия классов, структур, перечислений и т.д.), объявляемые пользователем.

Согласно правилам языка C#, при объявлении идентификаторов можно использовать алфавитно-цифровые символы и символ подчёркивания (нижний слеш). Начинаться идентификатор должен с буквы или символа подчёркивания. Не допустимо начинать идентификатор с цифры. Ниже приведены примеры нескольких идентификаторов:

```
int SomeVar;        //допустимый идентификатор
int somevar;        //допустимый идентификатор
int _SomeVar;       //допустимый идентификатор
int SomeVar2;       //допустимый идентификатор
int 3_SomeVar;      //не допустимый идентификатор
```

Язык C# регистро-зависим. Поэтому «SomeVar» и «somevar», будут восприниматься как различные идентификаторы.

Само собой разуметься, что не разрешается использовать ключевые слова в качестве идентификаторов, однако можно использовать ключевые и зарезервированные слова, предварив их символом «@». Например:

```
int @int;           //допустимый идентификатор
@int = 5;           //допустимый идентификатор
Console.WriteLine(@int); //выводит в консоль: 5
```

Интересно, что в приведённом примере идентификатором является слово «int», а символ «@» игнорируется, играя роль сигнализатора, который указывает, что в данном контексте ключевое слово меняет своё значение.

В рамках платформы .NET наряду с правилами именования существует понятие о стиле именования. Таким образом, для проектов .NET предлагается использовать три основных подхода к именованию переменных, в частности, и всей массы идентификаторов, в целом:

- **Pascal case convention** (или просто нотация Паскаля) предлагает начинать каждое отдельное слово в идентификаторе с символа в Верхнем регистре. Разделители между словами не используются. Начинаться идентификатор так же должен с символа в верхнем регистре. В идентификаторах предлагается использовать только алфавитные (буквенные) символы латинского алфавита.

  Ниже приведён пример объявления идентификаторов в **Pascal case convention**:

```
double SupplierPrice = 136.54;
/*азвание переменной состоит из двух английских слов:
- Supplier – поставщик
- Price – цена
Из чего вдальнейшем будет понятно, что имееться в виду «закупочаня» цена*/
double SuplementaryPrice = SupplierPrice * 0.2; // Suplementary – добавочная
double SellingPrice = SupplierPrice + SupplementaryPrice; // Selling – продажа
```

- **Camel case convention** идентичен **Pascal case**, но с одним отличием – начинается идентификатор с символа в нижнем регистре, а все последующие отдельные,

составляющие идентификатор, слова начинаются с символа в верхнем регистре.. В спецификации .NET Framework в параграфе «**указания к именованию**» (**Naming Guidelines**) рекомендуется использовать этот стиль именования при объявлении аргументов методов. Однако, это не означает, что в подобных случаях обязательно использовать именно такой стиль именования. Каждый, из перечисленных стилей, именования, «имеет равные права».

Ниже приведён пример, аналогичный предыдущему, с использованием идентификаторов в стиле **Camel case convention**:

```
double supplierPrice = 136.54;
/*азвание переменной состоит из двух английских слов:
- Supplier – поставщик
- Price – цена
Из чего вдальнейшем будет понятно, что имееться в виду «закупочаня» цена*/
double suplementaryPrice = supplierPrice * 0.2; // Suplementary – добавочная
double sellingPrice = supplierPrice + supplementaryPrice; // Selling – продажа
```

- **Uppercase convention** – провозглашает, что все символы идентификатора должны находиться в верхнем регистре. Такой стиль именования используют в тех случаях, когда идентификатор соответствует некоторым аббревиатуре или акроним у.

## Область видимости переменных

Область видимости переменной – это та часть кода, в пределах которой переменная доступна для использования. Областью видимости переменной является **блок**, в котором она объявлена. Блок начинается открывающей, а заканчивается закрывающей фигурными скобками.

```
{    //начало блока
     //тело блока
}    //конец блока
```

Блок может находиться внутри другого блока. В таком случае вводятся понятия внешнего и внутреннего блока соответственно. Пример вложенного блока приведён ниже:

```
      static void Main(string[] args)
      {     //начало внешнего блока, который одновременно является и телом метода

            {     //начало внутреннего блока
                  //тело внутреннего блока
            }     //конец внутреннего блока

       }     //начало внешнего блока
```

Переменные, объявленные во внешнем блоке, будут видны из внутреннего, но не наоборот. Локальные переменные, объявленные во внутреннем блоке, будут не видны из внешнего блока. Поскольку, «область видимости распространяется вовнутрь, но не наружу». Этот процесс иллюстрируется на следующем примере:

```
static void Main(string[] args)
{//начало внешнего блока
     { //начало внутреннего блока
         int i = 0;
     } //конец внутреннего блока
     int counter = 0;
     for (; i < 10; i++)/*на этой строке компилятор возвращает ошибку:
                          the name i does not exist in the current context
                         (мия i не существует в текущем контексте)*/
     {
         counter += i;  //переменная counter, напротив, видна во внутреннем блоке
     }
}//начало внешнего блока
```

Внутри блока переменная доступна всему коду, который указан после её объявления. Например, переменная, указанная вначале метода, будет видна всему коду метода. Тогда как переменная, объявленная в конце блока, становиться бесполезной, ввиду её недоступности.

**«Время жизни переменной»** начинается с момента её объявления и заканчивается закрывающей фигурной скобкой блока, в котором она была объявлена.

## 8. Ввод-вывод в консольном приложении

Одним из основных вопросов, возникающих при реализации приложений, является обеспечение диалога с пользователем. Такой диалог направлен на предоставление пользователю информации о выполнении программы, с одной стороны, и на

предоставление пользователю возможности управлять выполнением приложения, с другой.

При создании консольного приложения весь «пользовательский диалог» реализуется в текстовом режиме (пользователь получает от приложения информацию в текстовом виде и либо осуществляет выбор действия, либо вводит, необходимы для работы приложения, данные). Такое взаимодействие с пользователем заключается в реализации консольного ввода-вывода информации.

В системе базовых типов .NET Framework предусмотрен класс **Console,** который содержит набор статических **методов** и **свойств**, необходимых для осуществления консольного ввода-вывода, и получения служебной информации о **консоли**. Только что было упомянуто новое для Вас понятие – **«свойство».** Это понятие будет подробно рассматриваться в соответствующем уроке. Пока достаточно представлять свойство, как поле класса (компонентные данных класса), поскольку использование свойств не отличается от использования полей.

В классе Console определены следующие свойства:

- **BackgroundColor** – возвращает или устанавливает фоновый цвет выводимого в консоль текста (возвращает объект перечисления ConsoleColor).
- **BufferHeight** – возвращает или устанавливает высоту буферной зоны.
- **BufferWidth** – возвращает или устанавливает ширину буферной зоны.
- **CapsLock** – возвращает true, если нажата клавиша CapsLock.
- **CursorLeft** – возвращает или устанавливает номер колонки буферной зоны, в которой находиться курсор.
- **CursorSize** – возвращает или устанавливает высоту курсора относительно высоты ячейки символа.
- **CursorTop** – возвращает или устанавливает номер ряда буферной зоны, в которой находиться курсор.

- **CursorVisible** – возвращает или устанавливает значение индикатора видимости курсораю.
- **Error** – возвращает стандартный поток для вывода информации о возникающих ошибках (релевантен cerr в C++).
- **ForegroundColor** – возвращает или устанавливает цвет выводимого в консоль текста (возвращает объект перечисления ConsoleColor).
- **In** – возвращает стандартный поток ввода.
- **InputEncoding** – возвращает или устанавливает значение кодировки текста, которую консоль использует для чтения вводимой информации.
- **KeyAvailable** – возвращает true, если в стандартном потоке ввода доступна реакция на нажатие клавиш клавиатуры.
- **LargestWindowHeight** – возвращает наибольшее количество рядов в буферной зоне консоли, базируясь на значениях текущего шрифта и разрешения экрана.
- **LargestWindowWidth** – возвращает наибольшее количество колонок в буферной зоне консоли, базируясь на значениях текущего шрифта и разрешения экрана.
- **NumberLock** – возвращает true, если нажат NUM LOCK/
- **Out** – возвращает стандартный поток вывода.
- **OutputEncoding** – возвращает или устанавливает значение кодировки текста, которую консоль использует для форматирования выводимой информации.
- **Title** – возвращает или устанавливает текст заголовка окна консоли.
- **TreatConsoleCAsInput** – возвращает индикатор того, как используется комбинация клавиш Ctrl+C: является комбинацией прерывающей выполнение текущего действия в консоли (обрабатываемой системой) или передаётся в стандартный поток ввода.
- **WindowHeight** – возвращает или устанавливает ширину окна консоли.
- **WindowLeft** – возвращает или устанавливает отступ окна консоли слева, относительно экрана.

- **WindowTop** – возвращает или устанавливает отступ окна консоли сверху, относительно экрана.
- **WindowWidth** – возвращает или устанавливает высоту окна консоли.

Далее будут перечислены статические методы класса **Console**, специфичные для консольного ввода-вывода:

- **Beep –** проигрывает звук указанной частоты на протяжении указанного времени**.**
- **Clear –** очищает буфер консоли и окно консоли от теста.
- **MoveBufferArea –** копирует указанную область буфера консоли в указанную позицию.
- **OpenStandardError –** открывает стандартный поток вывода ошибок с указанным размером буфера.
- **OpenStandardInput –** открывает стандартный поток ввода с указанным размером буфера.
- **OpenStandardOutput –** открывает стандартный поток вывода с указанным размером буфера.
- **Read –** читает следующий символ из стандартного потока ввода.
- **ReadKey –** получает информацию о нажатой пользователем клавише (объект класса **ConsoleKeyInfo**).
- **ReadLine –** возвращает следующую строку текста из стандартного потока ввода.
- **ResetColor –** устанавливает значение цвета текста и фона в значение по умолчанию.
- **SetBufferSize –** устанавливает высоту и ширину буфера консоли.
- **SetCursorPosition** – устанавливает позицию курсора.
- **SetError –** передаёт объект класса **System.IO.TextWriter,** указанный в качестве параметра, в качестве значения свойства **Error.**
- **SetIn –** передаёт объект класса **System.IO.TextReader,** указанный в качестве параметра, в качестве значения свойства **In.**

- **SetOut –** передаёт объект класса **System.IO.TextWriter,** указанный в качестве параметра, в качестве значения свойства **Out.**
- **SetWindowPosition –** устанавливает позицию окна консоли относительно экрана.
- **SetWindowSize –** устанавливает размер окна консоли.
- **Write –** осуществляет вывод информации в стандартный поток вывода.
- **WriteLine –** аналогично методу **Write**, за исключением того, что данный метод дополняет выводимую строку служебным символом «**\n**» (переводит текст на следующую строку).

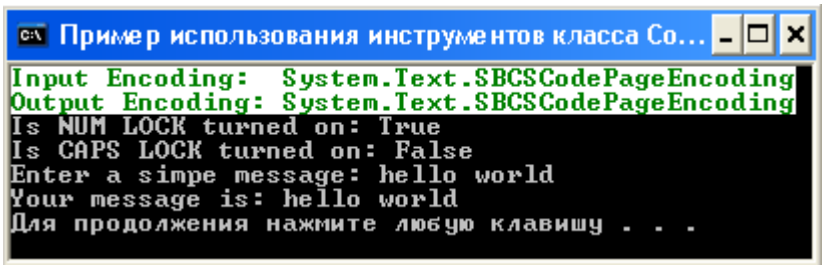
Следующий пример демонстрирует использование некоторых из описанных выше методов и свойств:

```
using System;

namespace ConsoleInputOutput
{
    class Program
    {
        static void Main(string[] args)
        {
            //изменяет заголовок окна консоли
            Console.Title = «Пример использования инструментов класса Console»;
            //изменяет цвет фона
            Console.BackgroundColor = ConsoleColor.White;
            //изменяет цвет текста
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            //получаем размер самого длинного сообщения в рамках нашей программы
            int length = (“Input Encoding:  “ +
                                Console.InputEncoding.ToString()).Length+1;
            //усанавливаем размер окна консоли
            Console.SetWindowSize(length, 8);
            /*усанавливаем размер буфера консоли
            (размер окна должен быть соответствующим и должен
            быть установлен до того,
            как мы изменим размер бьуфера)*/
            Console.SetBufferSize(length, 8);
            //выводим информацию о кодировке потока ввода
            Console.WriteLine(“Input Encoding:  “ +
                                Console.InputEncoding.ToString());
            //выводим информацию о кодировке потока вывода
            Console.WriteLine(“Output Encoding: “ +
                                Console.OutputEncoding.ToString());
            //устанавливает зеначение цвета фона и текста в значение по умолчанию
            Console.ResetColor();
            //выводим информацию о том, нажат ли NUM LOCK
            Console.WriteLine(“Is NUM LOCK turned on: “ +
                                Console.NumberLock.ToString());
            //выводим информацию о том, нажат ли CAPS LOCK
```

```
            Console.WriteLine("Is CAPS LOCK turned on: " +
                                    Console.CapsLock.ToString());
            /*выводим пользователю сообщение о том, что программа ожидает ввода
            некоторой информации*/
            Console.Write("Enter a simpe message: ");
            //получаем от пользователя текстовое сообщение
            string message = Console.ReadLine();
            //выводим сообщение, введённое пользователем
            Console.WriteLine(«Your message is: « + message);
        }
    }
}
```

Описанная выше программа формирует следующий вывод в консоль:




Как Вы можете видеть, цвет фона и текста первых двух строк изменён, а в окне консоли нет полос прокрутки, что означает совпадение значений размера буфера консоли и окна. Так же изменён текст заголовка окна консоли на указанное нами сообщение.

## 9. Структурные и ссылочные типы

Как уже упоминалось выше, в C# определены две категории типов данных:

- **Структурные типы данных** или **типы значений (иногда употребляется название – значащие типы)** (**value-types**)
- **Ссылочные** (**referenced – types**).

К типам значений относятся все объекты структур, а к ссылочным – объекты классов. Переменная значащего типа возвращает дубликат объекта, с которым связана, тогда как переменная ссылочного типа возвращает ссылку на объект, с которым связана. Отличие состоит и в размещении объектов в памяти: объекты значащих типов размещаются в стеке целиком, тогда как переменная ссылочного типа, размещённая в

стеке, хранит только ссылку на объект, который в действительности расположен в «куче» (основной массе оперативной памяти, выделяемой для хранения относительно больших объёмов данных).

Таким образом, если мы объявим переменную значащего типа “X” со стартовым значением 10 и переменную “Y” – равную “X”, а потом изменим значение “X“, то значение “Y” останется неизменным, поскольку при присваивании значение переменной “X” было сдублировано. Этот процесс демонстрируется на следующем примере:

```
using System;

namespace ValueAndReferencedTypes
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            int y = x;
            x = x – 5;
            Console.WriteLine(“X:\t” + x.ToString());
            //выводит в консоль: X: 5
            Console.WriteLine(“Y:\t” + y.ToString());
            //выводит в консоль: Y: 10
        }
    }
}
```

С переменными ссылочных типов дело обстоит иначе. Если мы объявим класс Point, который содержит два поля “X” и “Y”, и создадим переменную “a”, полю “X” которого присвоим значение 10. Потом создадим переменную “b” равную “a” и изменим поле “X” переменной “a”, то измениться и соответствующее поле переменной “b”, поскольку обе переменные ссылаются на один и тот же объект. Этот пример описан в нижеследующем листинге:

```
using System;

namespace ReferenceTypesSample
{
    class Program
    {
        class Point
        {
            public int X;
            public int Y;
        }
        static void Main(string[] args)
```

```
        {
            Point a = new Point();
            a.X = 10;
            Point b = a;
            b.X = b.X – 5;
            Console.WriteLine(“a.X:\t” + a.X.ToString());
            //выводит в консоль: a.X: 5
            Console.WriteLine(“b.X:\t” + b.X.ToString());
            //выводит в консоль: b.X: 5
        }
    }
}
```

Дело в том, что при объявлении переменной “b” нами не был создан новый объект. Инструкция создания нового объекта в C# выглядит следующим образом:

```
new SomeClass();
```

Поэтому, если бы мы хотели создать два различных объекта класса Point, то нам следовало бы поступить следующим образом:

```
using System;

namespace ReferenceTypesSample2
{
    class Program
    {
        class Point
        {
            public int X;
            public int Y;
        }
        static void Main(string[] args)
        {
            Point a = new Point();
            Point b = new Point();
            b.X = a.X = 10;
            b.X = b.X – 5;
            Console.WriteLine(“a.X:\t” + a.X.ToString());
            //выводит в консоль: a.X: 10
            Console.WriteLine(“b.X:\t” + b.X.ToString());
            //выводит в консоль: b.X: 5
        }
    }
}
```

# 10. Преобразование типов

Преобразование типов (приведение типов) – это процесс перевода значения объекта из одного типа данных в другой. Отличают две формы приведения типов:

- **неявное** (**implicit**) приведение (компилятор самостоятельно определяет – к какому типу данных необходимо привести значение).
- **явное** (**explicit**) приведение (тип, к которому нужно привести значение, «явно» указан разработчиком).

  Существуют правила приведения типов. Не все типы данных приводимы друг к другу.

### Неявное преобразование

Необходимо понимать, что при неявном преобразовании, значение будет приведено к более точному типу данных. То есть неявное приведение между совместимыми типами возможно только в том случае, когда оно происходит без потери информации. Например, тип **double** является более точными, нежели тип **float** (у типа **double** большее число разрядов дробной части). Таким образом, приведение типа **float** к типу **double** заключается в отсечении той части дробного числа, хранение которой не позволяет тип данных **float**. Конечно, возможна ситуация, при которой в переменной типа **double** храниться число с нулевой дробной частью. В таком случае никакой потери информации (даже при приведении к типу **int**) не произойдёт. Однако, мы не можем знать на этапе компиляции, какие значения будут принимать переменные на этапе выполнения программы. Поэтому компилятор запрещает такое неявное приведение, при котором потенциально возможна потеря информации.

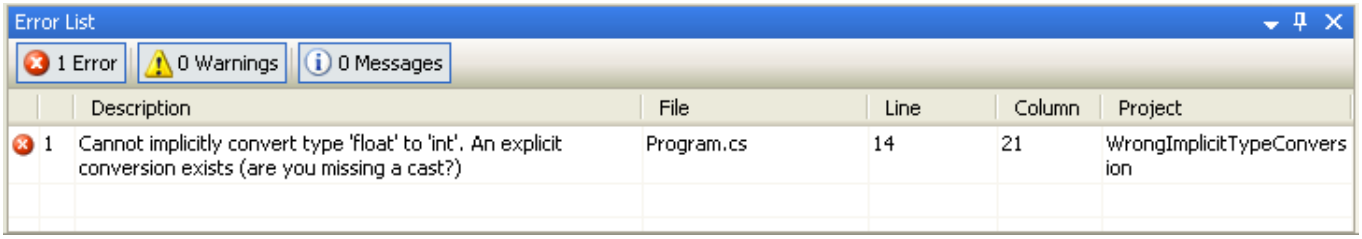
Использование неявного приведения типов показано на приведённом примере:

```
int x = 5;
/*
тип double не может быть неявно рприведён к типу float,
поскольку тип double более точный, то процедура приведения типа
происходит с протерейй информации,
поэтому необходимо указывать суфикс F
после инициализирующего значения
(о суфиксах говорилось в разделе 6 текущего урока)
*/
float y = 6.5F;
/*
значение переменной x (имеющей тип int)
неявно приводится к более точному типу float
*/
float b = y + x;
```

На следующем примере показана ошибка неявного преобразования типов:

```
float x = 6.5F;
int y = 5;
int A = y + x;
```

Этот пример возвращает следующую ошибку компилятора (“Canno’t implicitly convert type ‘float’ to type ‘int’.”; «Не могу неявно привести тип ‘float’ к типу ’int’»):




Для выполнения  указанной операции без ошибки, необходимо, чтобы переменная **“A”** имела тип данных **float**, как на примере ниже:

```
float x = 6.5F;
int y = 5;
float A = y + x;
```

Неявно допустимо приводить другу к другу следующие типы данных:

| Из типа | К типу |
|---|---|
| **byte** | short, ushort, int, uint, long, ulong, float, double, decimal |
| **sbyte** | short, int, long, float, double, decimal |
| **short** | int, long, float, double, decimal |
| **ushort** | int, uint, long, ulong, float, double, decimal |
| **int** | long, float, double, decimal |
| **uint** | long, ulong, float, double, decimal |
| **long** | float, double, decimal |
| **ulong** | float, double, decimal |
| **char** | ushort, int, uint, long, ulong, float, double, decimal |
| **float** | double |

### Явное преобразование

Для явного приведения типа необходимо указать этот тип данных в скобках непосредственно перед переменной или выражением:

```
double x = 5.7;
double y = 6.4;
/*выполняется явное приведение значения переменной,
имеющей тип данных double, к типу int*/
int A = (int)x;
/*выполняется явное приведение результата выражения,
имеющего тип данных double, к типу int*/
int B = (int)(x + y);
```

Допустимо явное приведение друг к другу следующих базовых типов данных:

| Из типа | К типу |
|---|---|
| **byte** | Sbyte или char |
| **sbyte** | byte, ushort, uint, ulong, char |
| **short** | sbyte, byte, ushort, uint, ulong, char |
| **ushort** | sbyte, byte, short, char |
| **int** | sbyte, byte, short, ushort, uint, ulong, char |
| **uint** | sbyte, byte, short, ushort, int, char |
| **long** | sbyte, byte, short, ushort, int, uint, ulong, char |
| **ulong** | sbyte, byte, short, ushort, int, uint, long, char |
| **char** | sbyte, byte, short |
| **float** | sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal |
| **double** | sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal |
| **decimal** | sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double |

В тех случаях, когда невозможно привести типы данных друг к другу (например: у Вас храниться число в строковом виде), можно использовать «конвертирование типов данных». Основные методы конвертирования инкапсулированы в статическом классе Convert. Методы тоже объявлены как статические, поэтому для использования необходимого метода не нужно создавать объект класса Convert. Использование некоторых методов показано на следующем примере: мы получаем из консоли строку, которая, предположительно, содержит строковую интерпретацию целого числа (примечание: необходимо вводить только числовые символы, поскольку пример не предусматривает проверки вводимых данных на корректность; попытка конвертирования в число нечисловых символов приведёт к ошибке времени выполнения). Доля конвертирования строки мы используем метод **ToInt32** класса **Convert**, который

возвращает тип данных **int**. То есть мы получаем числовую интерпретацию, введённых нами строковых данных. Листинг примера указан ниже:

```
//выводим пользователю сообщение о том,
//что необходимо ввести целое число в консоль
Console.Write(«Введите целое число: «);
//получаем строку из консоли в строковую переменную
string numberString = Console.ReadLine();
//конвертируем строковое значение в числовое
int number = Convert.ToInt32(numberString);
//выводим результат
Console.WriteLine(«Строка была успешно отконвертирована в тип данных int!»);
Console.WriteLine(«Число = « + number);
```

Согласно официальной документации, указанные ниже методы класса **Convert** реализуют следующее поведение:

- **ToBase64CharArray** – Преобразует значение подмножества массива 8-битовых целых чисел без знака в эквивалентное подмножество массива знаков Юникода, состоящее из цифр в кодировке **Base64**.
- **ToBase64String** – Преобразует значение массива 8-битовых целых чисел без знака в эквивалентное представление в виде значения типа **String**, состоящее из цифр в кодировке **Base64**.
- **ToBoolean** – Преобразует заданное значение в эквивалентное логическое значение.
- **ToByte** – Преобразует заданное значение в 8-битовое целое число без знака.
- **ToChar** – Преобразует заданное значение в символ Юникода.
- **ToDateTime** - Преобразует заданное значение к типу **DateTime**.
- **ToDecimal** – Преобразует заданное значение в число типа **Decimal**.
- **ToDouble** – Преобразует заданное значение в число двойной точности с плавающей запятой.
- **ToInt16** - Преобразует заданное значение в 16-битовое целое число со знаком.
- **ToInt32** – Преобразует заданное значение в 32-битовое целое число со знаком.
- **ToInt64** – Преобразует заданное значение в 64-битовое целое число со знаком.
- **ToSByte** – Преобразует заданное значение в 8-битовое целое число со знаком.

- **ToSingle** – Преобразует заданное значение в число одинарной точности с плавающей запятой.
- **ToString** – Преобразует заданное значение в эквивалентное представление в виде значения типа **String**.
- **ToUInt16** – Преобразует заданное значение в 16-битовое целое число без знака.
- **ToUInt32** – Преобразует заданное значение в 32-битовое целое число без знака.
- **ToUInt64** – Преобразует заданное значение в 64-битовое целое число без знака.

## 11. Операторы

Оператор — это символ, указывающий операцию, выполняемую над одним или несколькими аргументами.

Хотя синтаксис применения операторов несколько отличен от вызова методов, оператор это специальная функция, принимающая один или несколько операндов и возвращающая значение, записанная как символическое обозначение некоторого действия, выполняемого с операндами. Теоретически, встречая символическое значение операторов, компилятор вызывает, в зависимости от количества и типа операндов, соответствующую функцию, которая и производит необходимое действие с операндами или операндом (однако рефлектор этого не показывает). Например, арифметический оператор *****, умножающий два целых числа представлен функцией:

```
public static int operator *(int operand1, int operand2);
```

Подробнее на операторных функциях мы остановимся при изучении перегрузки операторов.

Все операторы C# делятся на классы:

1. Арифметические операторы.
2. Операторы отношений.
3. Логические операторы.
4. Битовые операторы.
5. Операторы присваивания.

По количеству принимаемых операндов операторы подразделяются на унарные, бинарные или тернарные. Операнды — это значения, которые оператор использует в качестве входных данных, могут быть допустимым выражением любого размера, и могут состоять из любого числа других операций

Унарные – получают на вход один операнд. Бинарные – принимают два операнда и, наконец, тернарные (а такой в C# всего один) – принимают три операнда.

Некоторые операторы ведут себя по-разному в зависимости от типа или количества переданных им операндов. Например, оператор сложения (+) ведет себя по-разному в зависимости от типа и количества переданных ему операндов.

Рассмотрим случай разного поведения оператора сложения в зависимости от типа переданных ему операндов. Если оба операнда являются числами, оператор сложения возвращает сумму этих значений. Если оба операнда являются строками, оператор сложения возвращает последовательное соединение двух операндов. В следующем программном коде показано, как оператор по-разному ведет себя в зависимости от операндов.

```
Console.WriteLine(5 + 5); // 10
Console.WriteLine(«5» + «5»); // 55
```

Поведение операторов также может отличаться в зависимости от числа передаваемых операндов. Оператор вычитание (-) является одновременно унарным и бинарным оператором. Если оператор вычитание используется только с одним операндом, оператор меняет его знак и возвращает результат.

Если оператор вычитания используется с двумя операндами, он возвращает разность между ними. В следующем примере показан оператор вычитания, используемый вначале как унарный оператор (отрицание), а затем как бинарный.

```
Console.WriteLine(-3); // -3
Console.WriteLine(7 – 2); // 5
```

Рассмотрим все операторы C# по категориям.

## Арифметические операторы

Арифметические операторы – используются для вычислений так же как в алгебре, то есть для выполнения основных арифметических действий и, в свою очередь, делятся на следующие группы:

| Группа | | Оператор | Выполняемая операция |
|---|---|---|---|
| Бинарные | Мультипликативные | * | Умножение |
| | | / | Деление |
| | | % | Остаток от деления |
| | Аддитивные | + | Сложение |
| | | - | Вычитание |
| Унарные | | + | Унарный плюс |
| | | - | Унарный минус (Отрицание) |
| | | ++ | Инкремент |
| | | -- | Декремент |

Как и в C++ операторы ++ и -- имеют две формы префиксную и постфиксную.

Постфиксные ++ и – берут какой-либо операнд и либо инкрементально увеличивают, либо декрементально уменьшают его значение. Хотя эти операторы и

являются унарными, они классифицируются отдельно от остальных унарных операторов из-за своего высокого старшинства и особого поведения. Если постфиксный оператор используется как часть более крупного выражения, значение выражения возвращается до обработки этого оператора.

Префиксные операторы инкремента и декремента, имеют меньший приоритет и отличаются от своих постфиксных эквивалентов тем, что операции инкрементального увеличения и декрементального уменьшения выполняются перед возвращением значения всего выражения.

### Операторы отношений

Операторы отношений – бинарные операторы, получают два операнда, сравнивают их значения, а затем возвращают логическое значение (True or False), используются для постановки условия в задаче.

| Оператор | Выполняемая операция |
|---|---|
| = = | Равно |
| != | Не равно |
| > | Больше |
| < | Меньше |
| >= | Больше или равно |
| <= | Меньше или равно |

Операторы отношения = = и != применимы к любому объекту, тогда как операторы >, <, >=, <= только к числовым типам и типам, которые поддерживают упорядывачивание.

Приведем небольшой пример использования операторов отношения

```
Console.WriteLine(2 > 7); //false
Console.WriteLine(2 != 4);// true
Console.WriteLine(8 < 10);//true
Console.WriteLine("my" == "my");//true
Console.WriteLine(false == true);//false
// Console.WriteLine(true > false); //ошибка!!!
```

**Логические операторы**

Логические операторы – операторы, реализующие операции булевой алгебры – сравнивают два логических типа и возвращают результат сравнения (true or false). Логические операторы предназначены для выполнения самых распространенных логических операций и перечислены в следующей таблице.

| Оператор | Выполняемая операция |
|---|---|
| && | Сокращенное И |
| \|\| | Сокращенное ИЛИ |
| & | И |
| \| | ИЛИ |
| ! | НЕ (унарный) |

Сокращенные формы операторов И и ИЛИ отличаются только тем, что при использовании сокращенной формы вычисление значения второго операнда происходит лишь в случае необходимости.

```
int a = 0;
Console.WriteLine(2 > 7 && 5 != 8); //false второй операнд
//не  вычисляется, так как результатом в первом
//операнде уже является false
Console.WriteLine(2 != 4 && 2 != 5);// true
Console.WriteLine(8 < 10 & 8 < 20);//true
```

```
Console.WriteLine("my“ =” "my“ |” "my“ !” "he“lo");”/true
Console.WriteLine(false == true);//false
Console.WriteLine(2 > 7 && 2 / a != 5);
/*нет ошибки!!! Второй операнд
      не вычисляется, так как результатом в первом
      операнде уже является false
      Console.WriteLine(2>7 & 2/a!=5);
      ошибка!!! Вычисляются оба
         операнда, причем вычисление второго приводит к ошибке*/
```

Ниже представлена таблица истинности для логических операторов:

| | |
|---|---|
| **&&, &** | **true && true = true;**<br>**true && false = false;**<br>**false && true = false;**<br>**false && false = false;** |
| **\|\|, \|** | **true \|\| true = true;**<br>**true \|\|false = true;**<br>**false \|\|true = true;**<br>**false \|\|false = false;** |

**Битовые операторы**

Битовые или поразрядные операторы – определены для целых числовых типов данных. С помощью битовых операторов можно проверять и модифицировать состояние отдельных битов соответствующих значений. В таблице приведена сводка таких операторов. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

| Оператор | Выполняемая операция |
|---|---|
| **&** | **Поразрядное И (AND)** |
| **^** | **Поразрядное исключающее ИЛИ (XOR)** |
| **\|** | **Поразрядное включающее ИЛИ (OR)** |
| **>>** | **Побитовый сдвиг вправо (деление)** |
| **<<** | **Побитовый сдвиг влево (умножение)** |
| **~ (унарный)** | **Битовое отрицание (NOT)** |

| Таблица истинности для И (&) | | | Таблица истинности для включающего ИЛИ ( \| ) | | | Таблица истинности для исключающего ИЛИ (^) | | |
|---|---|---|---|---|---|---|---|---|
| 1 операнд | 1 операнд | Результирующий бит | 1 операнд | 1 операнд | Результирующий бит | 1 операнд | 1 операнд | Результирующий бит |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Операторы побитового сдвига получают два операнда и сдвигают биты первого операнда влево или вправо на величину, указанную во втором операнде.

```
int a = 10;
int b = 1;
int result = a >> b;//деление на 2 в степени второго
//операнда, в данном случае в степени 1, то есть просто на 2
Console.WriteLine(result); // 10/2=5
result = a << b;// умножение 2 в степени второго
//операнда, в данном случае в степени 1, то есть просто на 2
Console.WriteLine(result); //10*2=20
result = a | 5;
Console.WriteLine(result); //15
result = a & 3;
Console.WriteLine(result);//2
result = a ^ 6;
     Console.WriteLine(result);//12
```

## Операторы присваивания

И последняя группа – операторы присваивания.

Эта группа включает в себя оператор присваивания равно, а также так называемые «составные» или «укороченные» операторы присваивания.

Оператор присваивания обозначается одинарным знаком равенства (=). Его роль в языке C# во многом такая же, как и в других языках программирования – он позволяет присвоить результат выражения или значение одного операнда другому. В C# оператор присваивания = позволяет создавать целую цепочку присвоений. Например:

```
int a, b, c;
a = b = c = 58;
```

Составные операторы присваивания упрощают написание кода и делают его более эффективным, так как вычисляют значение операнда только один раз. Составные или укороченные операторы присваивания существуют для всех бинарных операторов как арифметических, так и поразрядных.

| Оператор | Выполняемая операция |
|---|---|
| = | Равно |
| += | Сложение с присвоением |
| -= | Вычитание с присвоением |
| *= | Умножение с присвоение |
| /= | Деление с присвоением |
| %= | Присвоение остатка от деления |
| &= | Поразрядное И с присвоением |
| \|= | Включающее ИЛИ с присвоением |
| ^= | Исключающее ИЛИ с присвоением |

Приведем простой пример использования операторов присваивания:

```
int a = 10;
int b = 1;
int result = 0;
result = a + b;
Console.WriteLine(result); //11
result += b;
Console.WriteLine(result); //12
result -= a;
Console.WriteLine(result); //2
result *= 6;
Console.WriteLine(result);//12
result /= 3;
 Console.WriteLine(result);//4
```

**Приоритет операторов.**

Приоритет операторов определяет порядок, в соответствии с которым они будут выполнены. В C# операторы имеют следующий приоритет:

| |
|---|
| ***Высший*** |
| **++(постфиксный) - - (постфиксный)** |
| **! ~ + (унарный) – (унарный) ++ (префиксный) - - (префиксный)** |
| *** / %** |
| **+ -** |
| **>> <<** |
| **< > <= >=** |
| **= = !=** |
| **&** |
| **^** |
| **\|** |
| **&&** |
| **\|\|** |
| ***Низший*** |

Могут возникнуть ситуации, в которых в одном выражении встретятся два или большее число операторов одного старшинства. В таких случаях компилятор использует правила ассоциативности, чтобы определить, какой оператор обрабатывать в первую очередь.

Все бинарные операторы, за исключением оператора присваивания, являются ассоциируемыми слева, а это значит, что операторы обрабатываются слева направо.

Операторы присваивания и оператор условия (?:) являются ассоциируемыми справа операторами, и это означает, что операторы справа обрабатываются раньше, чем операторы слева.

Так что при написании кода обратите внимание на приоритет используемых вами операторов и, при необходимости, включите в выражение круглые скобки для прямого указания последовательности выполнения операторов.

Например, рассмотрим операторы + -, имеющие одинаковое старшинство и оператор *, имеющий больший приоритет.

```
int a = 10; int b = 1;
int result = a + b * 2;
Console.WriteLine(result); //12
result = (a + b) * 2;
Console.WriteLine(result); //22
result = a + b - 4 *-2;
Console.WriteLine(result); //3
result = (a + (b - 4))-* 2;
Console.WriteLine(result); //14
```

Итак, мы рассмотрели все виды операторов языка C#. Теперь вы можете сконструировать любое выражение с различными типами данных.

## 12. Условия

Условия – логические выражение, которые используется для реализации алгоритма ветвления. Они представляют собой выбор пути решения задачи в соответствии с

выполнением или невыполнением некоторого условия выбора. Условия выбора представляются в форме *условных выражений*.

*Простое* условное выражение - это–два выражения, между которыми помещается любой знак логического сравнения.

*Например:*

**a < 34** или *Sity ! = " "*

*С«о«ное* (составное) условное выражение - это–последовательность простых условий или других условных выражений, которые соединены между собой знаками логических операций.

*Например*:

**a > 2 && a < 10 (**значение переменной **a** больше 2 и меньше 10)

**a > 5 || b > 5 (**значение переменной **a** больше 5 или значение переменной **b** больше 5)

**!(a < 0) (**не верно, что значение переменной **a** меньше 0)

## Условный оператор if.

Условные операторы позволяют контролировать поток выполнения, исполняя те операции, которые согласуются с условием.

Условный оператор if используется для разветвления процесса обработки данных на два направления. Он может иметь одну из форм:

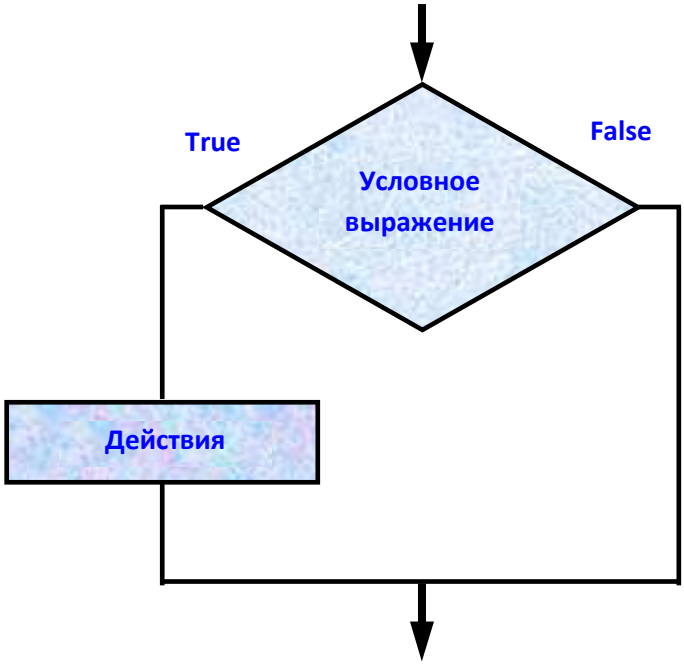
- сокращенную
- полную.

Форма сокращенного оператора if:

**if (**Условное Выражение) Действие;

*Логика работы такая же как в С++:*

- Если условное выражение имеет значение **true**, то делается переход к выполнению операторов (**Действия)**.
- Если условное выражение имеет значение **false**, выполнение условного оператора сразу заканчивается и начинают выполняться операторы программы, которые следуют за условным оператором.



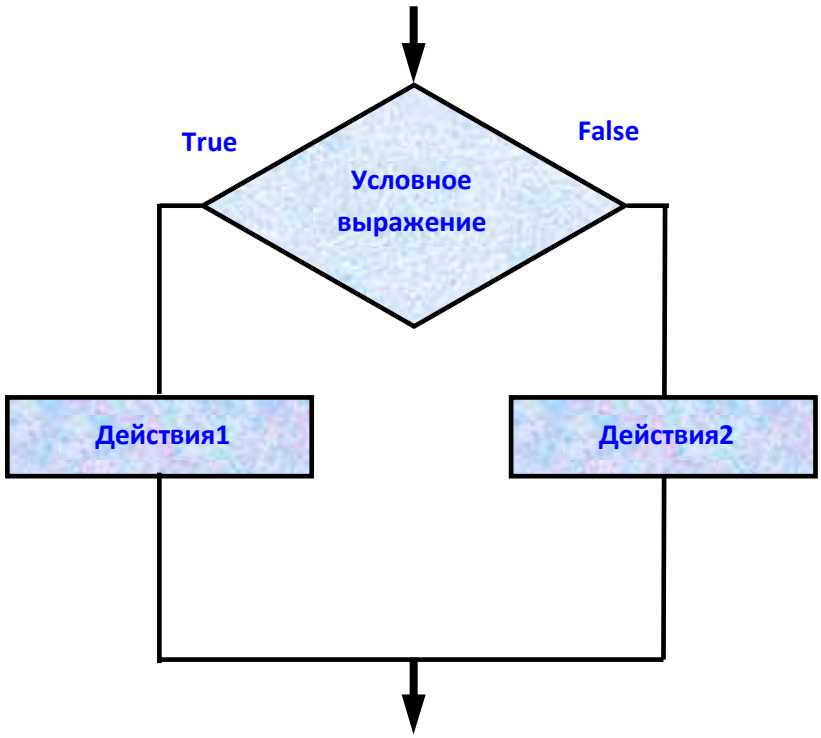

Например, простая программа, иллюстрирующая использования условного оператора **if**. Реализуем игру: пользователь загадывает число от 1 до 10. Это число сравнивается с числом сгенерированным компьютером и, если число угадано, пользователю выводится поздравление.

```
            int MaX = 10;
            Console.Write("Угад«йте число от 1 до {0}...", MA»);
            int userNumber = Convert.ToInt32(Console.ReadLine());
            Random rnd = new Random();
            double PcNumber = rnd.NextDouble() * MAX;
            PcNumber = Math.Round(PcNumber);
            Console.Write("Прав“льное число {0}, а вы задали {1}...\n",
”                                       PcNumber, userNumber);
            if (PcNumber == userNumber) // Число угадано!
            {
                Console.WriteLine("Позд«авляем!");
    »       }
```

## Условный оператор if else.

Позволяет выполнять код при выполнении определенного условия. Конструкция *else* оператора позволяет определить действия, которые нужно выполнить, если результатом вычисления выражения в *if* будет *false.*

```
if (УсловноеВыражение)
{
  //Дествия1;
}
else
{
  //Действия2;
}
```




Таким образом, с помощью полной формы оператора if можно выбрать одно из двух альтернативных действий процесса обработки данных.

***Рис 2.*** *Логика работы if- else*

Результатом вычисления выражения в операторе if должно быть булевское значение. Этим C# отличается от таких языков как C++, которые позволяют в операторе *if* сравнивать любые переменные на их совпадение с нулем. Этот пример показывает, какие распространенные ошибки могут быть допущены при использовании оператора *if в* C#:

```
namespace IfElseCondition
{
    class TestClass
    {
    } ;
    class Program
    {
        static void Main(string[] args)
        {
```

```
                int f = 1;
                TestClass my = new TestClass();
                if (f) // Ошибка: попытка перевести int в bool.
                {
                    Console.WriteLine(f);
                }
                if (my) // Ошибка: попытка перевести Test в bool.
                {
                    Console.WriteLine("Это «бьект класса Test");
 »              }
            }
        }
    }
```

Попытавшись скомпилировать этот код, вы получите такие сообщения об ошибках компилятора C#:

- Error 1 Cannot implicitly convert type 'int''to ’bool‘
- Err’r 2 Cannot implicitly convert type 'Cons‘leApplication1.TestClass' to ’bool‘

Ка’ видите, компилятор "прес«к" поп»тки использовать в операторе if выражения с небулевскими значениями. Причина в том, что в C# оператор *if* выполняет свою "есте«твенную" фун»цию — управляет ходом программы на основе результата логической проверки.

Ниже пример переписан так, чтобы компилятор не выдавал сообщений об ошибках. Каждая строка из предыдущей программы, вызывавшая ошибку, переписана так, чтобы выражение возвращало булевский результат.

```
namespace IfElseCondition2
{
    class TestClass
    {
    } ;
    class Program
    {
        static void Main(string[] args)
        {
            int f = 1;
            TestClass my = new TestClass();
            if (f > 0)
            {
                Console.WriteLine(f);
            }
            if (my != null)
            {
                Console.WriteLine("Это «бьект класса Test");
```

```
»           }
            else
            {
                Console.WriteLine("Test«);
»           }

        }
    }
}
```

**Условный оператор switch.**

Оператор switch является расширенным оператором ветвления, который позволяет в зависимости от значения выражения перейти к выполнению определенного кода.

В операторе *switch* вы указываете выражение, возвращающее некоторое значение и один или несколько фрагментов кода, которые будут выполняться в зависимости от результата выражения. Он аналогичен применению нескольких операторов *if/else,* но если в последних вы можете указать несколько условий (возможно, не связанных между собой), то в операторе *switch* содержится лишь один условный оператор, за которым следуют блоки, которые нужно выполнять. Вот его синтаксис:

```
switch (выражение)
{
  case константа 1:
     //блок операторов первой константы;
     break;
 [case константа 2:
     //блок операторов второй константы;
     break;
 ]
 [case константа n:
     //блок операторов n-й константы;
```

```
        break;
    ]
    [default:
        //операторы, выполняющиеся в том случае,
        //когда значение выражения не совпало
        //ни с одним из перечисленных значений констант;
        break;
    ]
}
```

*Выражение-переключатель* должно иметь тип *sbyte, byte, short, ushort, int, uint, long, ulong, char string* или же должно быть явно преобразовано в один из этих типов.

В каждом операторе *case* (кроме последнего блока) должен быть тот или иной *оператор_перехода,* включая оператор *break.*

По сути оператор *switch* работает так же, как if. Сначала вычисляется *выражение-переключатель,* а затем результат сравнивается со всеми *выражениями-константами* или *case-метками,* определенными в операторах *case.* При обнаружении совпадения управление передается первой строке кода в соответствующем операторе *case.*

Кроме нескольких операторов *case,* в *switch* можно указать оператор *default.* Это аналогично конструкции *else* в операторе if. Каждый оператор *switch* может иметь только одну метку *default.* При ее отсутствии, если значение *выражения-переключателя* не соответствует ни одной case-метке, управление передается на первую строку после закрывающей скобки оператора *switch.*

В отличие от языка C++, оператор switch языка C# не поддерживает передачу управления вниз (fall-through).

Обычно в C++ оператор *case* выполняется, когда *выражение-константа* совпадает с *выражением-переключателем.* Затем оператор *break* передает управление за пределы

оператора *switch.* Передача управления вниз означает, что при отсутствии *break* будет выполняться следующий оператор *case,* содержащийся в *switch.*

Передача управления вниз нужна, когда у вас две case-метки и вторая метка представляет операцию, которая должна выполняться в любом случае.

В C# это реализовано с помощью объединение case-меток, в этом случае нужно расположить case-метки одну за другой:

```
switch (выражение)
{
   case константа 1:
   case константа 2:
       //блок операторов второй константы;
      break;
  [case константа n:
       //блок операторов n-й константы;
       break;]
  [default:
       //операторы, выполняющиеся в том случае,
       //когда значение выражения не совпало
       //ни с одним из перечисленных значений констант;
       break;]
}
```

*Пример:*

```
int Num = 0, Price = 0;
switch (Num)
{
    //Если количество товара до 4 штук, то цена 25 Коп за 1 единицу
    //товара.
    case 1:
    case 2:
    case 3:
    case 4:
        Price = 25;
```

```
                break;
            //Если количество товара от 5 до 8 штук, то цена 23 Коп за 1
            //единицу товара.
            case 5:
            case 6:
            case 7:
            case 8:
                Price = 23;
                break;
            // Иначе устанавливаем цену в ноль.
            default:
                Price = 0;
                break;
        }
```

Оператор switch достаточно тесно связан с оператором goto, который имеет особый синтаксис (форму) употребления совместно с оператором switch, о чём будет подробно описано в разделе, посвящённом оператору goto.

**Оператор ?:.**

Условный оператор ***(?:)*** возвращает одно из двух значений в зависимости от значения логического выражения.

*Условие* **?** *выражение, вычисляемое в случае выполнения условия* **:** *выражение, вычисляемое в случае невыполнения условия*

Если условие имеет значение true, вычисляется и становится результатом первое выражение; если значение false, вычисляется и становится результатом второе выражение. В любом случае вычисляется только одно из двух выражений.

Пример

```
int myInt = 0;
int anotherInt = myInt != 0 ? 1 : 1234;
Console.WriteLine(anotherInt.ToString());
```

Условный оператор является правоассоциированным, поэтому выражение формы

a ? b : c ? d : e

вычисляется как

a ? b : (c ? d : e)


# 13. Циклы

*Цикл* - это–блок команд, который исполняется многократно заданное количество раз или до тех пор, пока не будет выполнено условие выхода из цикла.

С циклами тесно связаны такие понятия как тело цикла, итерация, счетчик, условие выхода.

Итак, последовательность инструкций, заключенная в блок и предназначенная для многократного исполнения, называется телом цикла, а однократное выполнение тела цикла называется итерацией.

Выражение определяющее, будет в очередной раз выполняться итерация, или цикл завершится, называется условием выхода или условием окончания цикла (либо условием продолжения в зависимости от того, как интерпретируется его истинность — как признак необходимости завершения или продолжения цикла).

Переменная, которая хранит текущий номер итерации, называется счётчиком итераций цикла или просто счётчиком цикла, причем цикл не обязательно содержит счётчик, счётчик не обязан быть один — условие выхода из цикла может зависеть от нескольких изменяемых в цикле переменных, а может определяться внешними условиями.

Исполнение любого цикла включает первоначальную инициализацию переменных цикла, проверку условия выхода, исполнение тела цикла и обновление переменной цикла на каждой итерации. Исходя из этого циклы подразделяются на:

- Циклы с предусловиями (сначала происходит проверка условия и только потом выполняется тело цикла)
- Циклы с постусловием (сначала выполняется тело цикла и только потом происходит проверка условия)

В C# к уже знакомым вам циклам for, while и do-while добавляется еще один: foreach – для работы с членами массивов, списков, коллекций.

## Цикл for

Цикл с предусловием. Смысловое предназначение - вып–лнять последовательность действий заданное количество раз.

Задается следующим образом:

```
for (инициализация_переменной; условие; выражение)
 {
      Действие;
 }
```

Если тело цикла составляет один оператор, то фигурные скобки можно опустить

```
for (i = 0; i ‹ 10; i++)
{
      /*Ваш код здесь*/
}
```

После ключевого слова for следует круглая скобка, внутри которой обязательно присутствуют три выражения, разделенные точкой с запятой:

1). Первое выражение, в нашем случае "i=0"«- о»е–ация, которую надо выполнить перед началом цикла. Зачастую в этом месте пишут декларацию переменной, которая существует только для цикла, тогда выражение выглядит, например так: "int « = 0". С »ем же успехом поле может быть пустым.

2). Второе выражение устанавливает условие, при котором цикл должен продолжаться - его–можно рассматривать как "Прод«лжать цикл пока выражение 2 истинно". Мо»ет быть пустым, но тогда вам самим придется внутри цикла проверять условие выхода и самим же выходить из цикла иначе цикл будет безусловным (бесконечным).

3). Третье выражение задает операцию, которую надо выполнить по завершении каждого круга цикла, обычно используется для приращения счетчика управляющего продолжением цикла. Тоже может быть пустым.

Итак, в строчке примера показан цикл, обнуляющий переменную i вначале, исполняющийся пока переменная i меньше 10, после каждого круга увеличивающий i на 1, что дает 10 выполнений цикла, при нормальных условиях.

Рассмотрим еще один пример: напишем программу определяющую, является ли заданная строка палиндромом.

**Цикл while.**

Цикл с предусловием, что соответствует стратегии: "снач«ла проверь, а потом делай". В »езультате проверки может оказаться, что тело такого цикла может ни разу не выполниться, используются в тех случаях, когда количество итераций заранее не известно

Задается следующим образом:

```
while (выражение)
{
        Действие;
}
```

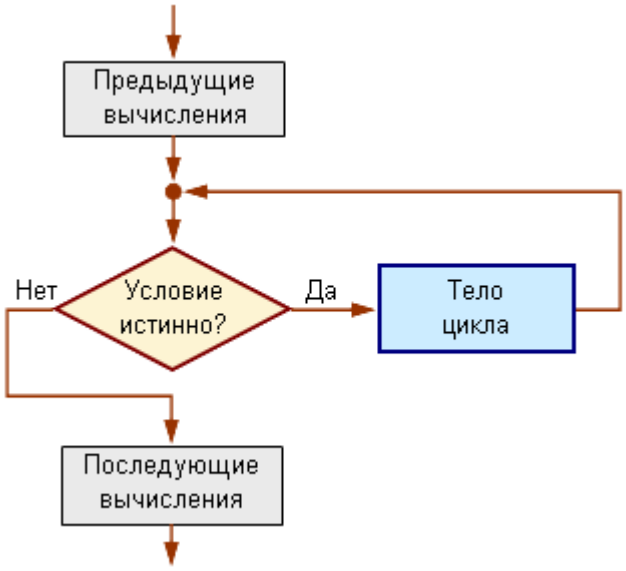
Если тело цикла составляет один оператор, то фигурные скобки можно опустить

Тело цикла выполняется до тех пор, пока остается истинным выражение, записанное после ключевого слова while.

На приведённом ниже рисунке представлена функциональная схема циклического оператора while:

Тело цикла выполняется до тех пор, пока остается истинным выражение, записанное после ключевого слова while.

На приведённом ниже рисунке представлена функциональная схема циклического оператора while:




```
// Сложение чисел сгенерированых
//компьютером в диапазоне от 0 до 20 пока сумма не будет
//равна 100, вывести количество чисел
int counter = 0; //счетчик для чисел
Random rand = new Random();
int number; //переменная для хранения числа
int summ = 0;  //переменная для хранения суммы
while (summ <= 100) //пока сумма меньше 100
{
    number = rand.Next(0, 20); //генерируем число
    summ += number; //добавляем к сумме
    counter++;       //прибавляем счетчик
}
Console.WriteLine(
      "Для «уммы было сгенерировано {0} чисел от 0 до 20", co»nter);
```

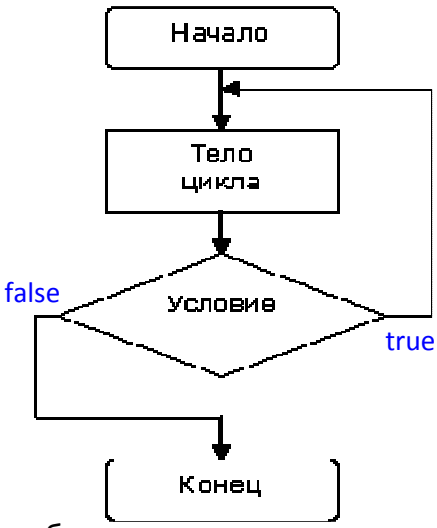
**Цикл do while.**

Цикл с постусловием, т.е. тело такого цикла выполняется, по меньшей мере, один раз.

Задается следующим образом:

```
do
{
    Действие;
}while(выражение);
```

Ниже представлен рисунок с функциональной схемой циклического оператора do while:




Тело цикла может быть одиночным оператором или блоком операторов, а условие управляет циклом, и может быть любым действительным логическим выражением. Тело цикла выполняется в том случае, если условие принимает значение *true*. Если же условие принимает значение *false*, управление программой передаётся строке кода, которая следует сразу за циклом.

```
            // посчитать сумму всех цифр любого числа
            int sum = 0; //переменная для суммы
            int N = 0; //любое число
            Console.WriteLine("Введ«те любое целое число:");
»           N = Convert.ToInt32(Console.ReadLine());
            int a = N; //Запоминаем число
            do
            {
                sum = sum + N % 10; //находим последнюю цифру и суммируем её
                N = (N - N %-10) / 10; //отсекаем последнюю цифру от числа
            } while (N > 0);
            //если N больше нуля, вернуться и повторить операторы тела цикла
                  Console.WriteLine("Сумм« всех цифр числа  {0} = {1}", a,»sum);
```


**Цикл foreach.**

Новым видом цикла, которого нет в C++, является цикл **foreach**, удобный при работе с массивами, коллекциями и другими подобными контейнерами данных. Его синтаксис:

```
foreach(тип_идентификатор in контейнер)
{
        Действие;
}
```

Цикл работает в полном соответствии со своим названием - тел– цикла выполняется для каждого элемента в контейнере. Тип идентификатора должен быть согласован с типом элементов, хранящихся в контейнере данных. На каждом шаге цикла идентификатор, задающий текущий элемент контейнера, получает значение очередного элемента в соответствии с порядком, установленным на элементах контейнера. С этим текущим элементом и выполняется тело цикла - вып–лняется столько раз, сколько элементов находится в контейнере. Цикл заканчивается, когда полностью перебраны все элементы контейнера.

Серьезным недостатком циклов foreach в языке C# является то, что цикл работает только на чтение, но не на запись элементов. Так что наполнять контейнер элементами приходится с помощью других операторов цикла.

В приведенном ниже примере показана работа с одномерным массивом. Массив создается с использованием циклов типа for, а при нахождении суммы его элементов, минимального и максимального значения используется цикл foreach.

*Примечание:*

*В данном примере для хранения информации используется массив. Массивы не рассматриваются в текущем уроке. Но семантика объявления и использования*

*массивов схожа с использованием массивов в C++, поэтому данный пример не должен составить трудности. Массивы будут подробно рассмотрены в последующих уроках.*

```
        /// Демонстрация цикла foreach. Вычисление суммы,
        /// максимального и минимального элементов
        /// одномерного массива, заполненного случайными числами.
        int[] arr3d = new int[10];
        Random rand = new Random();
        for (int i = 0; i < 10; i++)
        {
            arr3d[i] = rand.Next(100);
        }
        long sum = 0;
        int min = arr3d[0], max = arr3d[0];
        foreach (int item in arr3d)
        {
            sum += item;
            if (item > max)
                max = item;
            else if (item < min)
                min = item;
        }
        Console.WriteLine("summ“ = {0}, minimum = {1}, maximum = {2}",
    ”  sum, min, max);
```

**Инструкция break.**

Прерывает текущий вложенный цикл или условный оператор, в котором он присутствует. После этого управление передается на строку кода, следующую за встроенным оператором этого цикла или условного оператора. Оператор break указывается в том месте, откуда вы хотите передать управление и имеет простейшую форму, без скобок или аргументов: break;

В следующем примере приложение будет выводить все числа от 1 до 100, кратные 6. Но когда значение счетчика достигнет 66, break прервет цикл for.

```
        for (Int i = 1; i <= 100; i++)
        {
            if (i % 6 == 0)
            {
                Console.WriteLine(i);
            }
            if (i == 66)
            {
                break;
            }
        }
```

## Инструкция continue.

Как и **break**, оператор **continue** позволяет изменять выполнение цикла. Но **continue** не завершает встроенный оператор текущего цикла, а останавливает текущую итерацию и передает управление на начало цикла для следующей итерации.

В следующем примере проверяется, не повторяются ли строки в массиве. Один из вариантов решения — просматривать массив с помощью вложенных циклов, сравнивая один элемент с другим. Но следует учитывать, что, если один индекс массива (i) совпадает с другим индексом этого массива (i), это значит, что сверяется один и тот же элемент и сравнивать их не нужно. В этом случае используем оператор continue чтобы прекратить текущую итерацию и передать управление на начало цикла.

```
string[] words = { "один«, "д», «т»и"« "о»ин«, "т»",«"о»",«"с»м"«
      »       "вос"« "д»в"« "д»с"«};
»nt counter = 1;
int index = -1;
Console.WriteLine("Обра”отка массива...")I”r (int i = 0; i < 10; i++)
{
     for (int j = 0; j < 10; j++)
     {
          if (i == j) continue;
          if (i == index) i++;
          if (words[i] == words[j])
          {
                counter++;
                index = j;
          }
      }
      if (counter > 1)
           Console.WriteLine("'{0}”’вст’ечается в массиве {1} раза",
     ”words[i], counter);
       else
              Console.WriteLine("'{0}« в массиве встречается только 1 раз",
          »  words[i]);
       counter = 1;
}
```

Пример программы с использованием инструкций break и continue: Подсчет средней заработной платы сотрудников, причем текущая заработная плата должна превышать 1 000.

```
            double sr_zarplata=0;   // средняя заработная плата
            double zarplata=0; // текущая заработная плата
            int kol=0; // количество введенных данных для подсчета средней зарплаты
            do
            {
                Console.WriteLine("Введ«те зарплату \n для окончания введите” +
                                                                “значение -1) :\n");
»               zarplata=Convert.ToDouble(Console.ReadLine());
                if(zarplata < 0)
                    break;
                if(zarplata < 500)
                    continue;
                else
                {
                    sr_zarplata+=zarplata;
                    kol++;
                }
            }while   (1==1);
            sr_zarplata = sr_zarplata/kol;
            Console.WriteLine("Сред«яя зарплата = {0}", sr»zarplata);
```

**Инструкция goto.**

Goto - это– оператор безусловного перехода. Может иметь одну из следующих форм:

- goto ‘идентификатор’,
- goto case ‘выражение-константа’,
- goto default.

При применении первой формы нужно указать метку в том месте программы куда нужно осуществить переход. Метка указывается просто - в н–жном месте программы пишется имя метки с двоеточием на конце:

```
 Exit:  return 0;

      Чтобы перейти на эту метку:

goto Exit;
```

Если в текущем методе такой метки нет, при компиляции возникнет ошибка. Еще одно важное правило: goto может применяться для выхода из вложенного цикла. Однако если он находится вне области видимости метки, при компиляции возникнет ошибка. Так что перейти внутрь вложенного цикла невозможно.

Еще goto можно применять если нужно перейти к определенной ветви в блоке switch, это будет вторая и третья форма goto:

```
int level = Parse.ToInt32(Console.ReadLine());
switch (level)
{
        case 0:
                Console.WriteLine("Уров“нь 0");
        ”break;
        case 1:
                goto case 2;
        case 2:
                Console.WriteLine("Уровень от 1 до 2");
                goto default;
        default:
                Console.WriteLine("Пока");
                break;
}
```

Последняя форма оператора goto позволяет переходить на метку default в операторе switch, что дает возможность написать один блок кода, который будет выполнен в результате нескольких вычислений в switch.

## 14. Домашнее задание

1. Даны целые положительные числа A, B, C. Значение этих чисел программа должна запрашивать у пользователя. На прямоугольнике размера A * B размещено максимально возможное количество квадратов со стороной C. Квадраты не накладываются друг на друга. Найти количество квадратов, размещенных на прямоугольнике, а также площадь незанятой части прямоугольника.

   Необходимо предусмотреть служебные сообщения в случае, если в прямоугольнике нельзя разместить ни одного квадрата со стороной C (например, если значение C превышает размер сторон прямоугольника).
2. Начальный вклад в банке равен 1000 руб. Через каждый месяц размер вклада увеличивается на P процентов от имеющейся суммы (P — вещественное число, 0 < P < 25). Значение P программа должна получать у пользователя. По данному P определить, через сколько месяцев размер вклада превысит 1100 руб., и вывести найденное количество месяцев K (целое число) и итоговый размер вклада S (вещественное число).
3. Даны целые положительные числа A и B (A < B). Вывести все целые числа от A до B включительно; каждое число должно выводиться на новой строке; при этом каждое число должно выводиться количество раз, равное его значению(например, число 3 выводится 3 раза).

   Например: если A = 3, а B = 7, то программа должна сформировать в консоли следующий вывод:

```
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
```

4. Дано целое число N (> 0), найти число, полученное при прочтении числа N справа налево. Например, если было введено число 345, то программа должна вывести число 543.