

Платформа Microsoft .NET и язык программирования C#



Урок №1

Введение в платформу Microsoft.NET

Содержание

1. Введение в платформу Microsoft .NET	6
История и этапы развития технологий программирования	6
Причины возникновения платформы Microsoft .NET	11
Сравнительный анализ преимуществ и недостатков платформы Microsoft .NET	12
2. Базовые понятия платформы Microsoft .NET ...	16
Архитектура платформы Microsoft .NET	16
Общезыковая среда исполнения CLR (Common Language Runtime)	18
Общая система типов CTS (common type system)	19
Общая языковая спецификация CLS (common language specification)	20

Библиотека классов FCL (BCL)	21
Языки платформы Microsoft .NET.....	25
Язык CIL (Common Intermediate Language)	26
Схема компиляции и исполнения приложения платформы Microsoft .NET	26
Понятия метаданных, манифеста, сборки	28
3. Введение в язык программирования C#	31
Плюсы и минусы языка программирования C#	31
Простейшая программа на языке программирования C#	32
4. Рефлекторы и дотфускаторы	41
Что такое рефлектор?	41
Необходимость использования рефлектора.....	42
Обзор существующих рефлекторов	42
Что такое дотфускатор?	44
Необходимость использования дотфускаторов ..	46
Обзор существующих дотфускаторов	47
5. Типы данных.....	49
Целочисленные типы данных.....	49
Типы данных для чисел с плавающей точкой.....	51
Символьный тип данных.....	54
Логический тип данных.....	55
6. Nullable типы.....	56
Что такое nullable типы	56

Цели и задачи nullable типов.	57
Операции, доступные для nullable типов	57
Примеры использования	58
7. Литералы	59
8. Переменные.	63
Понятие переменной.	63
Правила именования переменных	65
Область видимости переменных.	68
9. Ввод-вывод в консольном приложении.	70
10. Структурные и ссылочные типы.	78
11. Преобразование типов	79
Неявное преобразование.	79
Явное преобразование	82
12. Операторы	87
Арифметические операторы.	90
Операторы отношений.	92
Логические операторы	93
Битовые операторы	95
Операторы присваивания	97
Приоритет операторов.	99
13. Условия	101
Условный оператор if	101
Условный оператор if else.	104
Условный оператор switch	106

Тернарный оператор ?:	108
14. Циклы	109
Цикл for	111
Цикл while	113
Цикл do while	115
Цикл foreach	116
Инструкция break	118
Инструкция continue.	118
Инструкция goto	120
15. Домашнее задание	122

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 развитие 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 «С».

Первым языком этого семейства был язык программирования «С», разработанный Дэнисом Ритчи в 1970-х годах. Как и все популярные языки программирования,

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

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

Язык «С» имел большой успех, в связи с тем, что в нём сочетались гибкость, мощьность и удобство. Поэтому новый язык «С++» стал «развитием» языка «С». Можно сказать, что «С++» — это объектно-ориентированный

«С», а причиной его возникновения стала «мода» на объектно-ориентированное программирование.

Тесное родство «С++» и «С» обеспечило популярность новому языку программирования, поскольку «С»-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 находится 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 базе своего нового языка «Oak»

(англ. Дуб), который впоследствии стал называться «Java». Авторство этого языка приписывают Джеймсу Гослингу. На базе языка Java была создана среда исполнения «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). 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 равносителен 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). Впоследствии был описан в стандарте ECMA-334.

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 (**межязыковая 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 также 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 (*Common Intermediate Language* — **CIL** 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 (библиотека базовых классов — *Base Class Library*) содержатся 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 (CLR)*), 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 относительным: поскольку получившийся в результате компиляции исполняемый файл сохраняется во временной директории, поэтому при последующих запусках при-

ложения повторная компиляция происходить не будет. 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, когда 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 платформы 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» языке 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*, общезыковая 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 метод возвращает значение, то все «ветки» кода, которые определены в этой методе тоже должны возвращать значение, другими словами, приложение не будет компилироваться, пока не будет уверенности в том, что метод во всех ситуациях возвращает некоторое значение).

.NET Framework Class Library (FCL) — универсальный набор классов, для использования в разработке программ. FCL, 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 термини-

нологии .NET Framework FCL так же называют **BCL** (англ. *Base Class Library* — Библиотека Базовых Классов).

На рисунке 2.1 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 общезыковой среды исполнения и библиотеки базовых классов с пользовательскими приложениями и операционной системой в целом:

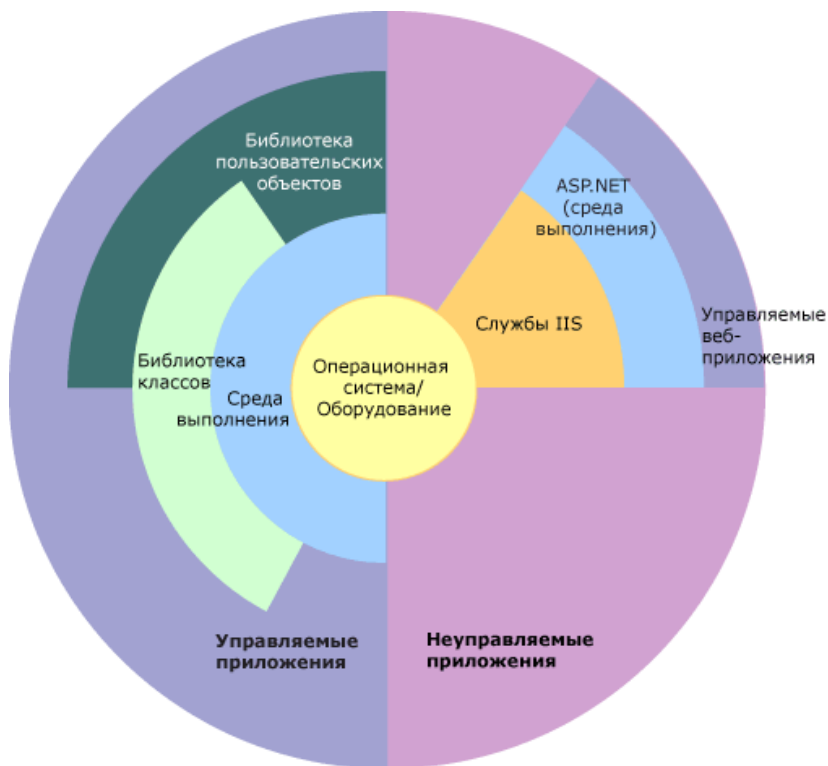


Рисунок 2.1. Обобщенная схема архитектуры платформы .NET Framework

Общезыковая среда исполнения CLR (Common Language Runtime)

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

В настоящий момент ведутся работы как минимум над еще двумя реализациями, этого стандарта — 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**).

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.

CLI описана в стандарте ECMA-335 (<http://www.ecma-international.org/publications/standards/Ecma-335.htm>) и основными составляющими её 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 двумерной 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, сериализованными 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 многопоточное 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 — C#, Visual C++ .Net (с управляемыми расширениями), 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, Perl, 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 CIL (Common Intermediate Language)

Общий промежуточный язык описан в третьем разделе стандарта ECMA-335 и является составной частью общей спецификации CLI, описанной выше. Компиляторы всех языков семейства .NET генерируют CIL на основании программного кода, тем самым обеспечивая интеграцию языков.

Команды на языке CIL чем-то напоминают команды на ассемблере, но включает некоторые высокотехнологичные конструкции, поэтому его еще называют «высокоуровневый ассемблер». Хотя писать код, используя непосредственно CIL и возможно, Вам вряд ли придется этим заниматься, но знать, как он выглядит не обходимо, мы обязательно к этому вернемся в разделе 3.

При чтении литературы или посещении различных форумов Вы, наверняка, помимо аббревиатуры CIL встретите еще два возможных варианта: **IL** (*Intermediate Language*), то есть просто «промежуточный язык» или **MSIL** (*Microsoft Intermediate Language*) это устаревшее название, которое было изменено после выхода стандарта ECMA-335.

Схема компиляции и исполнения приложения платформы Microsoft .NET

Схема компиляции и исполнения приложения платформы Microsoft .NET Framework изображена на рисунке 2.2.

Сам процесс выполнения программы происходит следующим образом. Компилятор одного из языков семейства .NET Framework на основании программного кода формирует исполняемый модуль, так называемый **Portable Executable (PE)** файл Windows с одним из расширений

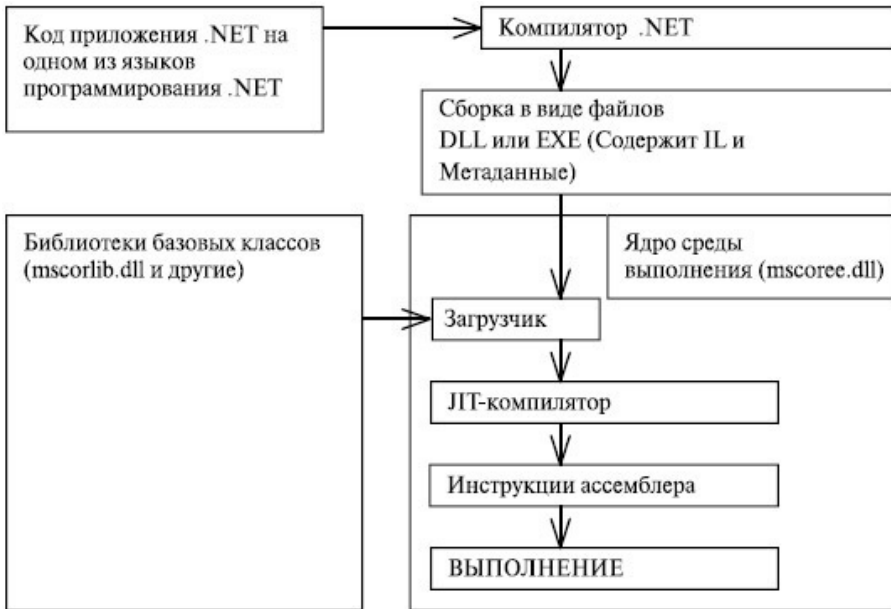


Рисунок 2.2. Схема компиляции и исполнения приложения платформы Microsoft .NET

(exe или dll). В данном файле содержится следующая информация:

- **заголовок PE** — показывает тип файла, время сборки файла, содержит сведения о выполняемом коде;
- **заголовок CLR** — содержит информацию для среды выполнения модуля (версию среды выполнения (Framework), характеристики метаданных, ресурсов и т.д.);
- **метаданные** — информация о типах, определенных в исходном коде (более подробно в следующем подразделе);
- **CIL** — промежуточный код, сгенерированный компилятором на основании исходного кода.

За выполнение исполняемого модуля на целевой ЭВМ «отвечает» CLR, которая физически на компьютере находится в виде файла `mscorlib.dll` и «знает» особенности конкретной ЭВМ. Первым делом CLR проверяет наличие на машине необходимой версии Framework, которая берется из заголовка CLR. Если необходимая версия отсутствует, то программа просто не запустится и будет выведено окно с сообщением об отсутствии конкретной версии Framework.

Далее в промежуточный код подключаются необходимые библиотеки из BCL (физически находится в виде файла `mscorlib.dll`), информация о которых содержится в метаданных.

После этого CLR активизирует JIT-компилятор (*Just In Time* — как раз вовремя), который по мере необходимости переводит CIL-код в команды процессора. При этом результаты деятельности JIT-компилятора сохраняются в оперативной памяти. Между фрагментом оттранслированного CIL-кода и соответствующим блоком памяти устанавливается соответствие, которое в дальнейшем позволяет CLR передавать управление командам процессора, записанным в этом блоке памяти, минуя повторное обращение к JIT-компилятору.

Понятия метаданных, манифеста, сборки

В данном разделе обсудим некоторые базовые понятия, которые необходимы для лучшего понимания специфики программирования на платформе .NET Framework.

При преобразовании программного кода приложения в промежуточный код, еще формируется и блок, так называемых **метаданных**, который содержит информацию

о данных, используемых в программе. Фактически это наборы таблиц, которые включают в себя информацию о типах данных, определяемых в модуле. Ранее такая информация сохранялась отдельно, теперь метаданные являются частью исполняемого модуля.

В частности, метаданные используются для:

- **сохранения информации о типах.** При компиляции теперь не требуются заголовочные и библиотечные файлы. Всю необходимую информацию компилятор читает непосредственно из управляемых модулей;
- **верификации (проверки)** кода в процессе выполнения модуля;
- **управления динамической памятью** (освобождение памяти) в процессе выполнения модуля;
- **обеспечения динамической подсказки (IntelliSense)** при разработке программы стандартными инструментальными средствами (Microsoft Visual Studio .NET) на основе метаданных.

Сборка (Assembly) — базовый строительный блок приложения в .NET Framework. Сборка является логической группировкой одного или нескольких управляемых модулей или файлов ресурсов. Управляемые модули в составе сборок исполняются в CLR. Сборка может быть либо исполняемым приложением (при этом она размещается в файле с расширением `.exe`), либо библиотечным модулем (в файле с расширением `.dll`). При этом ничего общего с обычными исполняемыми приложениями библиотечными модулями сборка не имеет.

Манифест (*Manifest*) или **Декларация сборки** — составная часть сборки. Это еще один набор таблиц метаданных, который:

- идентифицирует сборку в виде текстового имени, ее версию, культуру и цифровую сигнатуру (если сборка распределяется среди приложений);
- определяет входящие в состав файлы (по имени и хэшу);
- указывает типы и ресурсы, существующие в сборке, включая описание тех, которые экспортируются из сборки;
- перечисляет зависимости от других сборок;
- указывает набор прав, необходимых сборке для корректной работы.

3. Введение в язык программирования C#

Плюсы и минусы языка программирования C#

Первым делом опишем преимущества языка C#.

Язык программирования C# является полностью объектно-ориентированным языком программирования, то есть вся работа с кодом построена на объектах, также расширена поддержка событийно-ориентированного программирования. В отличие от C и C++, язык C# ориентирован на безопасный код (использование управляемой кучи при выделении памяти и т.д.). Язык C# является «родным» языком для создания приложений в среде Microsoft .NET, поскольку наиболее тесно и эффективно интегрирован с ней, при этом система типизации максимально близка к CTS.

Кроме того, язык программирования C# призван реализовать компонентно-ориентированный подход к программированию, который способствует меньшей машинно-архитектурной зависимости результирующего программного кода, большей гибкости, переносимости и легкости повторного использования (фрагментов) программ.

К недостаткам языка C# можно отнести его относительно невысокую производительность по сравнению с тем же кодом на языке C++.

В дальнейшем Вас ожидает изучение различных дисциплин с использованием языка C#, приведем их перечень:

- **Windows Forms** — технология создания клиентских приложений Windows;
- **Windows Presentation Foundation (WPF)** — технология создания клиентских приложений с гораздо большими возможностями и имеющими более насыщенный дизайн чем в Windows Forms;
- **ADO.NET** — технология, позволяющая работать с базами данных из приложений;
- **SP** — системное программирование, в ходе этой дисциплины изучается работа с потоками и все что с этим связано;
- **NP** — сетевое программирование — изучаются способы передачи информации по сети с использованием различных протоколов;
- **Windows Communication Foundation (WCF)** — сервис-ориентированная технология разработки приложений (создание и применение сервисов);
- **ASP.NET** — технология создания web-приложений.

Как Вы видите, общение с языком C# Вам предстоит длительное, поэтому имеет смысл выучить его, как можно лучше. Не будем медлить ...

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 (Integrated Development Environment* — Интегрированная среда раз-

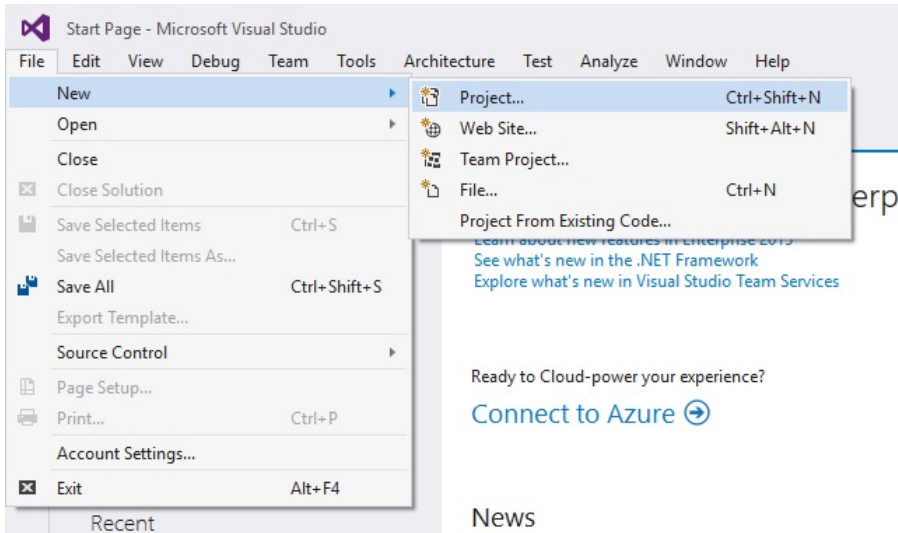


Рисунок 3.1. Создание нового проекта

работки) Вы будете использовать Microsoft Visual Studio 2015. Для создания консольного приложения в Visual Studio, в меню File необходимо в подменю New выбрать пункт Project (Рисунок 3.1). Иными словами мы говорим студии, что необходимо создать новый проект.

Как реакция на нажатие пункта меню появляется диалог создания проекта (Рисунок 3.2).

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.

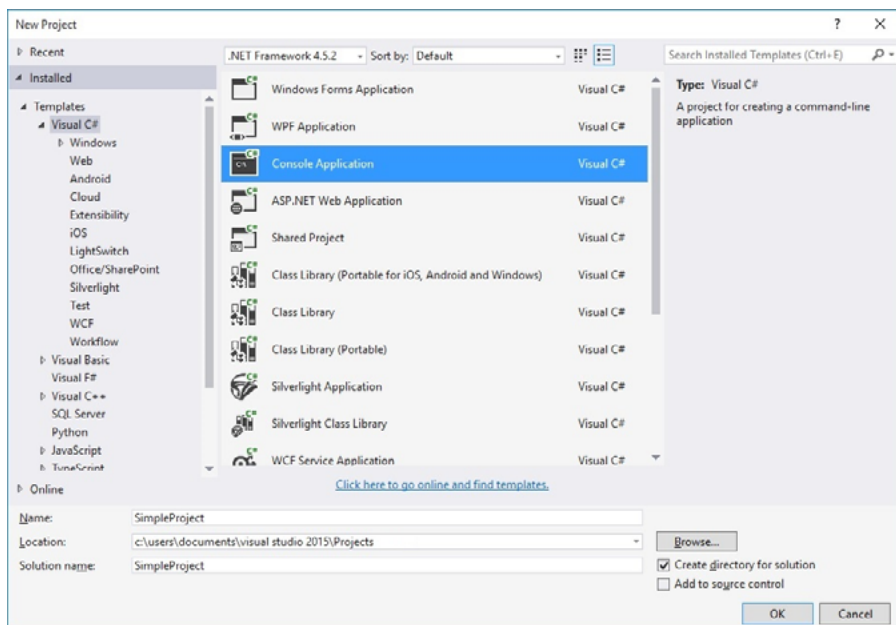


Рисунок 3.2. Создание консольного проекта

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 объявлен метод `Main`.

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 рисунке 3.3.

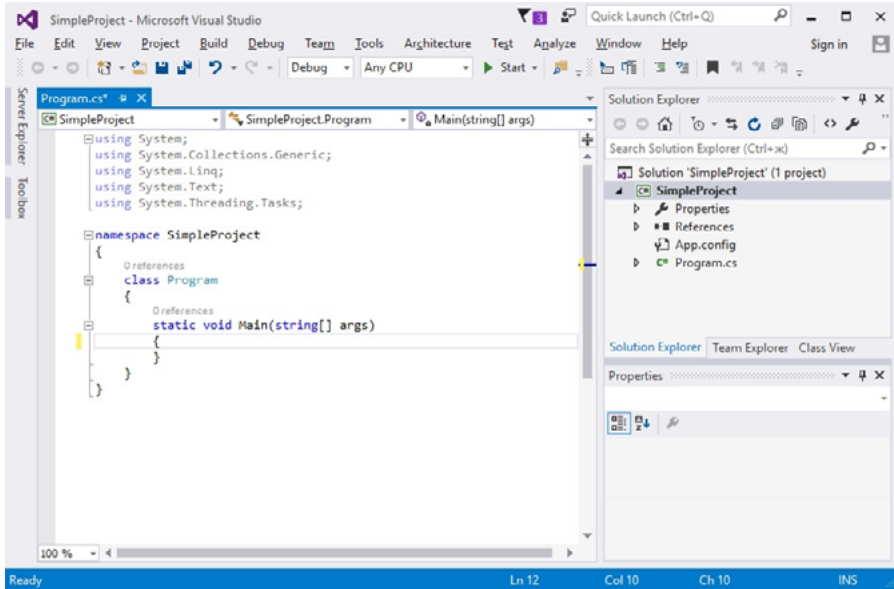


Рисунок 3.3. Первоначальный вид файла Program.cs

В тело метода 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 компиляции проекта в среде разработки Вам достаточно нажать комбинацию клавиш Ctrl+F5. После компиляции проекта сразу запуститься консоль, в которой будет выполняться Ваша программа.

Выполнить компиляцию программы можно также при помощи компилятора командной строки `csc.exe`. Для этого необходимо запустить инструмент Visual Studio 2015 — **Developer Command Prompt**. Вы можете найти его в меню «Пуск», как показано на рисунке 3.4:

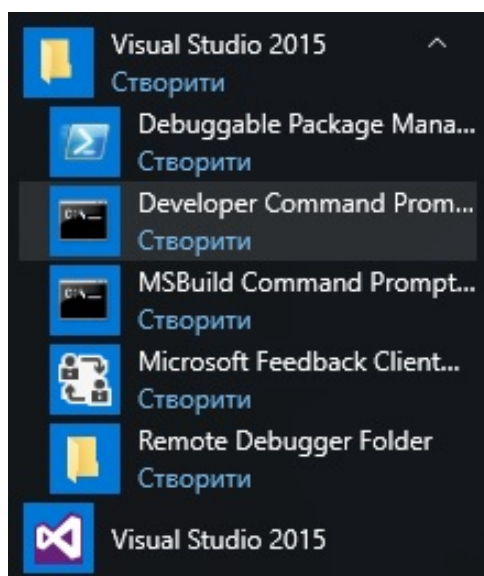


Рисунок 3.4. Запуск Developer Command Prompt

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

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

Рассмотрим выполненную нами команду по элементам.

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

1. /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);

2. /out: — указывает расположение результирующего файла;

Последним аргументом был указан файл с исходным текстом программы.

Результат выполнения команды показан на рисунке 3.5.

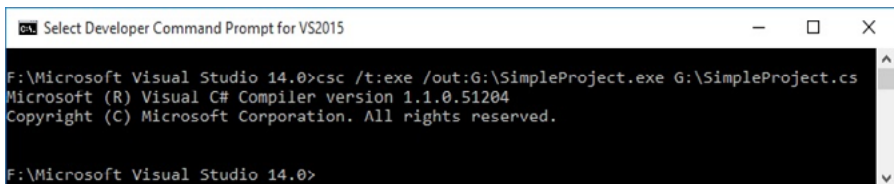


Рисунок 3.5. Результат выполнения команды

После выполнения данной команды в корневой директории диска «G:» появиться файл «SimpleProject.exe» (Рисунок 3.6).

Хотя файл SimpleProject имеет расширение .exe — он не содержит машинный исполняемый код. Как уже

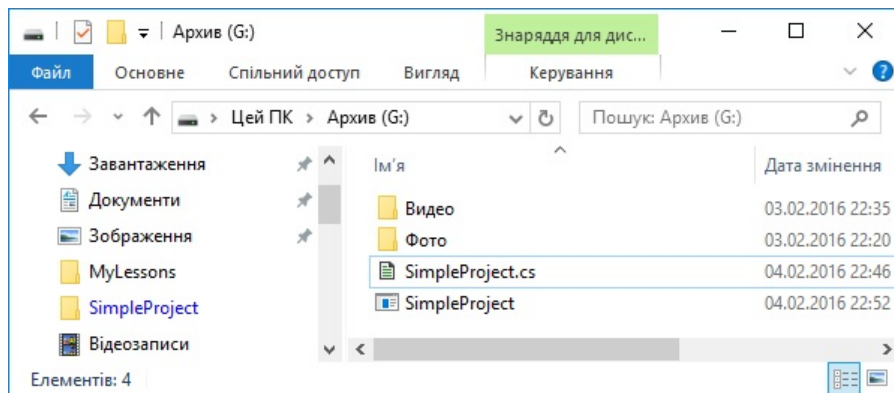


Рисунок 3.6. Полученный файл на промежуточном коде

упоминалось выше, все .NET-приложения компилируются в промежуточный код (CIL), который можно увидеть при помощи дизассемблера промежуточного языка Microsoft — ILDASM (*Microsoft Intermediate Language Disassembler*). Эта утилита отображает метаданные и инструкции языка CIL, связанные с соответствующим .NET-кодом и используется при отладке приложений. Для того чтобы запустить дизассемблер необходимо в командной строке ввести команду `ildasm` (Рисунок 3.7).

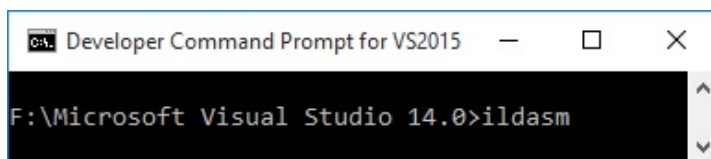


Рисунок 3.7. Команда ildasm

В появившемся окне выбираем пункты меню **File** — **Open** и указываем тот файл, промежуточный код которого хотим посмотреть (Рисунок 3.8).

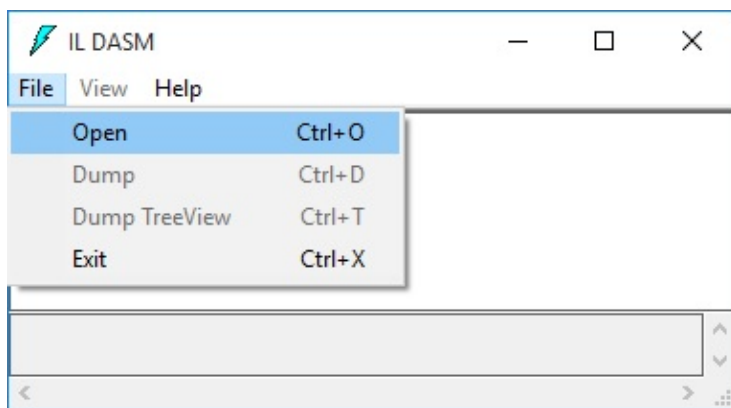


Рисунок 3.8. Открытие файла

После выбора файла в окне отображается структура проекта (методы, классы и т.д.), также отображается и манифест (Рисунок 3.9).

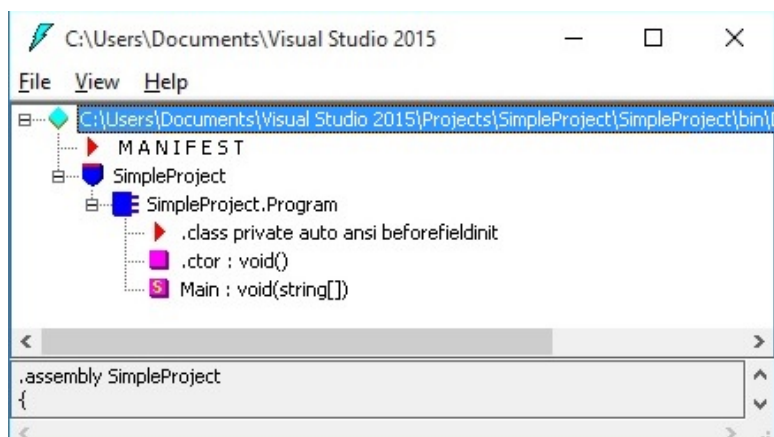


Рисунок 3.9. Структура проекта

Сам CIL код можно посмотреть, если выбрать любой из отображаемых пунктов. На рисунке 3.10 показан CIL код метода Main.

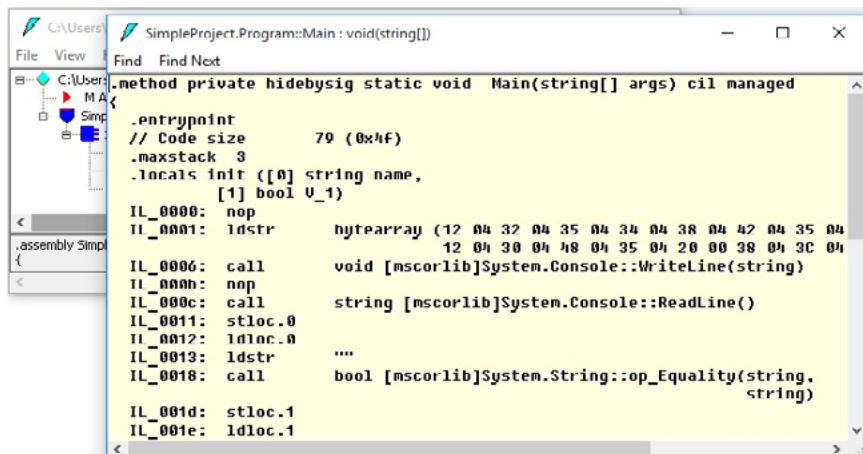


Рисунок 3.10. CIL код метода Main

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

```
csc -?
```

Результат этой команды показан на рисунке 3.11.

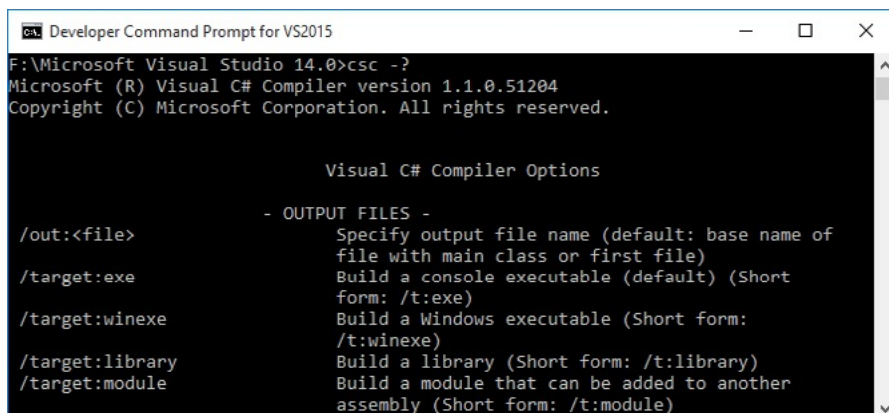


Рисунок 3.11. Аргументы компилятора csc

4. Рефлекторы и дотфускаторы

Что такое рефлектор?

Как упоминалось выше, проекты, созданные с использованием технологии **.NET**, компилируются не в исполняемый машинный код, а в CIL-код, который по сравнению с машинным кодом легче «поддаётся» декомпиляции (восстановлению оригинального текста программы из текста исполняемого файла). 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 связываясь с коллегами, для того, чтобы они передали Вам необходимый исходный файл — достаточно воспользоваться рефлектором.

Необходимость использования рефлексора

Использование рефлексора предоставляет возможность увидеть, как выглядит сборка после её компиляции. Так же, рефлексор 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), рефлексор может использоваться как средство анализа при тестировании продукта. Так же при наличии специальных дополнений рефлексор может позволять осуществлять анализ содержимого сборки на наличие дублирующих элементов кода и многое другое.

Обзор существующих рефлексоров

На сегодняшний день доступно определенное количество рефлексоров как платных, так и бесплатных, рассмотрим некоторые из них. Наиболее известным рефлексором является *.NET Reflector* от компании Red Gate's. Среди бесплатных альтернатив можно отметить *ILSpy* (<http://ilspy.net/>), *JetBrains dotPeek* (<http://www.jetbrains.com/decompiler/>) и *Telerik JustDecompile* (<http://www.telerik.com/products/decompiler.aspx>), принцип работы которых практически одинаков.

Мы рассмотрим действие рефлексора на примере приложения *JetBrains dotPeek*.

Для того чтобы посмотреть исходный код какой-либо программы необходимо в меню окна приложения выбрать пункты File/Open (Рисунок 4.1).

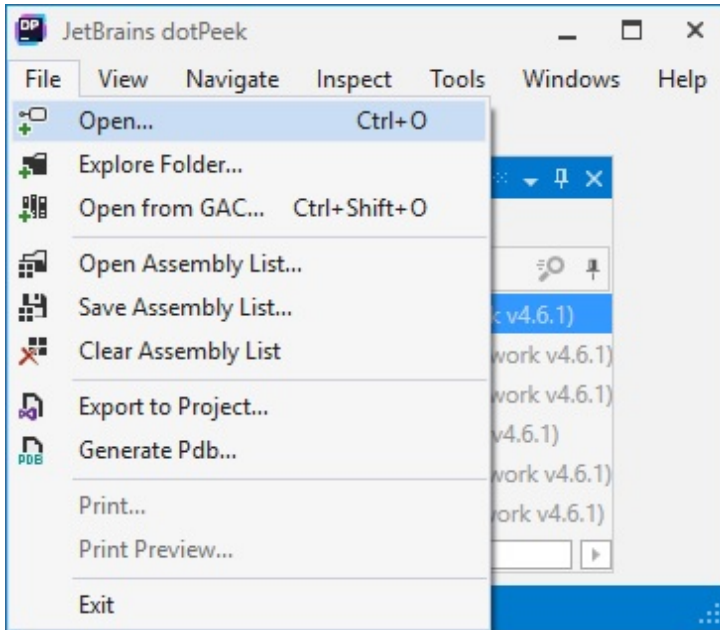


Рисунок 4.1. Открытие файла

После этого открывается диалоговое окно открытия файла (Рисунок 4.2). Найдите необходимый Вам файл на промежуточном коде с расширением `.exe` или `.dll`.

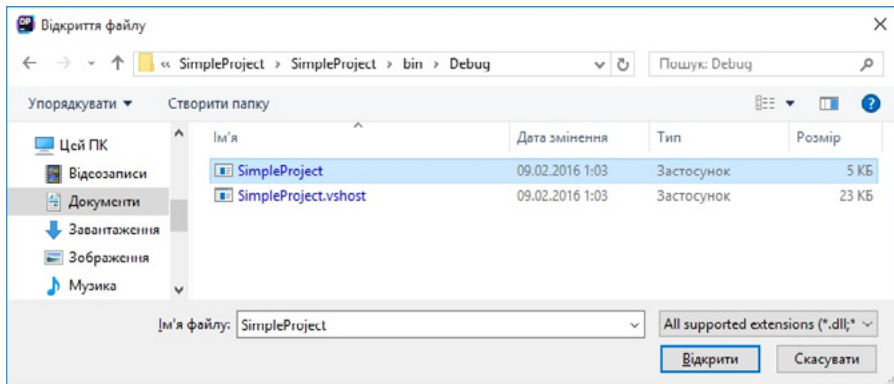


Рисунок 4.2. Выбор файла

После выбора нужного файла, в окне Assembly Explorer появиться название выбранного проекта и полная информация о нем. Двойной клик мыши по методу Main() проекта приведет к отображению кода метода в правой части окна (Рисунок 4.3).

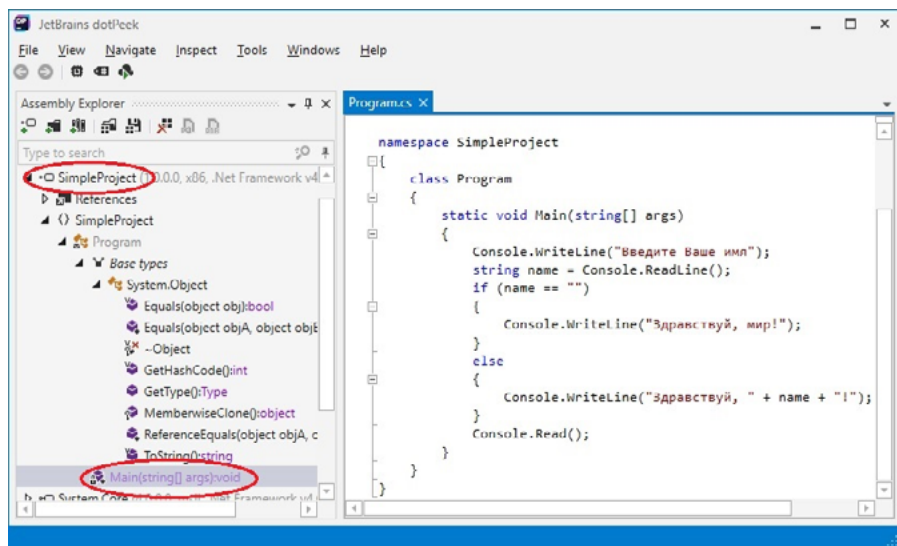


Рисунок 4.3. Результат работы программы

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 с одной стороны усложняет процесс обратной инженерии, 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 пиратами.

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 и находится в пункте PreEmptive Dotfuscator and Analytics меню Tools. Однако этот дотфускатор «умеет» только выполнять «переименование», что нельзя назвать надёжной защитой, например, если основным ноу-хау 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. ***Phoenix Protector*** — абсолютно бесплатный продукт компании NTCore. Список его возможностей состоит из обфускации, причем, есть возможность использовать встроенный список исключений, во избежание нежелательных результатов, и склеивания сборок. Несмотря на столь небольшой перечень возможно-

стей, механизм обфускации реализован достаточно умело, и его вполне можно использовать для базовой защиты продуктов .NET.

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 Formatter** — 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\$.

5. 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: значимые типы (*value-types*) и ссылочные типы (*reference-types*). Определено, что экземпляры значащих типов должны располагаться в стеке, тогда как ссылочных — в другой области оперативной памяти, называемой управляемой кучей (*managed heap*).

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 рисунке 5.1.

Для описания значимых 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

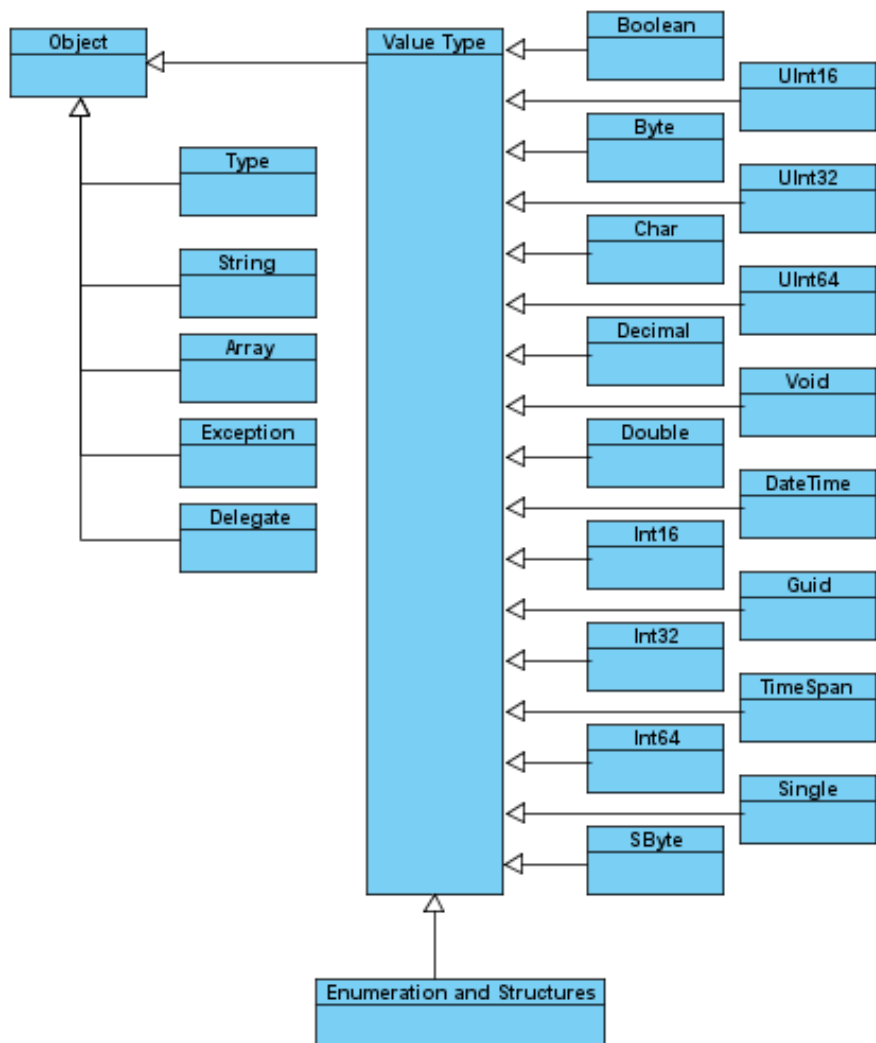


Рисунок 5.1. Общая иерархическая структура
Базовой Системы Типов

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 2 147 483 647
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 -1.797693e308 до 1.797693e308
decimal	Decimal	+	128	от -7.922816e28 до 7.922816e28

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 divisor = 3;
devidend = devidend / divisor;
//нижеследующая строка выводит в консоль
//0,333333333333333333333333333333
Console.WriteLine(devidend);
//нижеследующая строка выводит в консоль
//0,999999999999999999999999999999
//из чего можно сделать вывод, что ошибки
//округления привели к потере данных
```

```

Console.WriteLine(devidend * divisor);
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
//0,3333333333333333
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" («код-овая точка» или «код-овый 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 находится 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 бувнным
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

```

Логический тип данных

У логического типа могут быть только два типа значения — true (истина) или false (ложь). В C# этот тип задается при помощи ключевого слова bool, что соответствует типу System.Boolean. Используется при проверке каких-то условий (условный оператор, циклы). В отличие, от C++ в C# не определено взаимное преобразование логических и целых значений, например true не преобразуется в 1, а false не преобразуется в 0.

6. Nullable типы

Что такое nullable типы

Как говорилось выше, все числовые типы данных являются значимыми типами, им никогда не присваивается значение `null`, поскольку оно служит для установки пустой ссылки на объект.

На следующем примере попытка присвоить переменным "number" значение `null`,

```
double number = null; // Ошибка!  
string str = null; // запись корректна
```

приведет к ошибке на этапе компиляции: "Cannot convert null to 'double' because it is a non-nullable value type" (Не удастся преобразовать `null` в `double`, потому что это не nullable тип).

Nullable тип — это тип данных, который помимо значений, лежащего в его основе типа данных, может принимать еще и значение `null`.

Чтобы объявить переменную nullable типа, необходимо добавить к имени лежащего в основе типа данных вопросительный знак. Например:

```
int? nullInt = null;  
nullInt = 10;  
bool? nullBool = true;  
nullBool = null;  
//string? str = null; // ошибка на этапе  
компиляции
```


Если применить такой синтаксис к типу данных `string`, то это приведет к ошибке на этапе компиляции.

Синтаксис объявления nullable типов является на самом деле сокращенной формой создания экземпляра обобщенного типа структуры `System.Nullable<T>`. Хотя знакомство с обобщениями у Вас произойдет в дальнейших уроках, представим предыдущий пример в новом виде.

```
Nullable<int> nullInt = null;  
nullInt = 10;  
Nullable<bool> nullBool = true;  
nullBool = null;
```

Цели и задачи nullable типов

Nullable типы особенно полезны при работе с базами данных, поскольку значения в отдельных полях таблиц могут быть неопределенны (`null`). Вот как раз в таких случаях очень удобно использовать nullable типы.

Операции, доступные для nullable типов

Помимо операций, свойственных лежащему в основе типу данных для nullable типов существует еще и специальная операция `??`. Эта операция позволяет присваивать переменной любого типа данных определенное значение, в том случае если текущее значение nullable типа равно `null`.

```
int? nullInt = null;  
nullInt = nullInt ?? 50;  
Console.WriteLine(nullInt); // 50  
  
int number = nullInt ?? 100;  
Console.WriteLine(number); // 50
```

Операция ?? является сокращенной формой записи проверки значения переменной с использованием условного оператора.

```
int? nullInt = null;  
if (nullInt == null)  
{  
    nullInt = 100;  
}  
Console.WriteLine(nullInt); // 100
```

Примеры использования

Примеры использования nullable типов будут рассмотрены нами при работе с информацией из баз данных в рамках предмета ADO.NET.

7. 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 понадобится 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, литерал:

- объявленный с суффиксом «**L**» или «**l**» будет иметь тип `long`;

- с суффиксом «F» или «f» будет иметь тип float;
- с суффиксом «D» или «d» будет иметь тип double;
- с суффиксом «M» или «m» будет иметь тип decimal;
- суффикс «U» или «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 шестнадцатеричное число 0xFF соответствует
десятичному числу 255*/
```

В отдельную группу литералов выделены управляющие символьные последовательности, которые не имеют символического 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 текста 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*/

```

8. Переменные

Понятие переменной

Переменная — это именованный объект, хранящий значение некоторого типа данных. Язык С# относится к «*type-safe*» языкам. Иными словами, компилятор С# (C# *compiler*) гарантирует нам, что расположенное в переменной значение всегда будет одного и того же типа. Так как С# 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 С# переменная должна быть обязательно инициализирована перед использованием.

ПРИМЕЧАНИЕ:

В спецификации языка С# 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 на этапе **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 8.1.

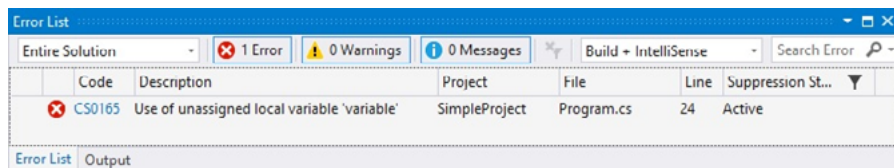


Рисунок 8.1. Ошибка на этапе компиляции

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 правилах именования, подразумевают не только имена переменных, но и 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. Недопустимо 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# регистрозависим, поэтому «SomeVar» и «somevar», будут восприниматься как различные идентификаторы.

Само собой разумеется, 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;
/*название переменной состоит из двух английских слов:
- Supplier — поставщик
- Price — цена
```

```

Из чего в дальнейшем будет понятно, что имеется
в виду «закупочная» цена*/
double SupplementaryPrice = SupplierPrice * 0.2; //
Supplementary — 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;
/*название переменной состоит из двух английских
слов:
- Supplier — поставщик
- Price — цена
Из чего в дальнейшем будет понятно, что имеется
в виду «закупочная» цена*/
double supplementaryPrice = supplierPrice * 0.2; //
Supplementary — 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 акрониму.

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
}
```

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

}
```

Переменные, объявленные во внешнем блоке, будут видны из внутреннего, но не наоборот. Локальные пе-

ременные, 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
                        (имя 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, становится бесполезной, ввиду её недоступности.

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

9. 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, необходимые 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**». Свойства класса — это методы двунаправленного доступа к полям класса, которые обеспечивают инкапсуляцию внутри типа внутренней логики доступа к данным (свойства также называют методами- аксессорами — от английского "access", то есть

«доступ»). Мы не будем подробно останавливаться на свойствах, поскольку они являются темой отдельного урока. На данном этапе достаточно подразумевать под свойствами класса соответствующие им поля, тем более что «внешне» обращение к свойствам ничем не отличается от обращения к полям.

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 находится курсор.
- **CursorSize** — возвращает или устанавливает высоту курсора относительно высоты ячейки символа.
- **CursorTop** — возвращает или устанавливает номер ряда буферной зоны, в которой находится курсор.
- **CursorVisible** — возвращает или устанавливает значение индикатора видимости курсора.
- **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.
- **TreatControlCAsInput** — 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 текста.
- **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 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";
            Console.BackgroundColor = ConsoleColor.
                White; //изменяет цвет фона
            //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;

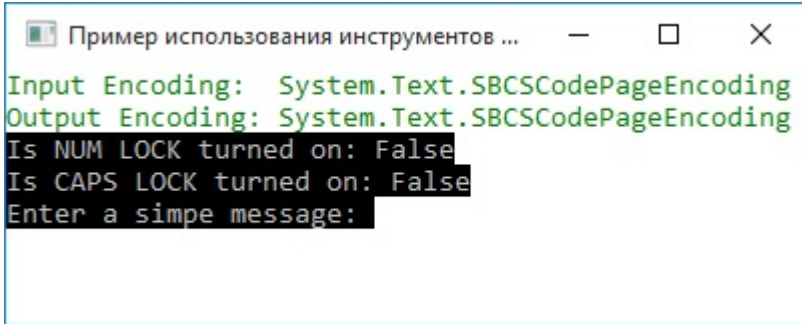
            Console.SetWindowSize(length, 8);
            //устанавливаем размер окна консоли
            /*устанавливаем размер буфера консоли
            (размер окна должен быть
            соответствующим и должен быть
            установлен до того, как мы изменим
            размер буфера)*/
            Console.SetBufferSize(length, 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 значение цвета 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 результат (Рисунок 9.1).



```
Пример использования инструментов ...
Input Encoding: System.Text.SBCSCodePageEncoding
Output Encoding: System.Text.SBCSCodePageEncoding
Is NUM LOCK turned on: False
Is CAPS LOCK turned on: False
Enter a simpe message:
```

Рисунок 9.1. Результат работы программы

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.

10. Структурные и ссылочные типы

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

- *Структурные типы данных или значимые типы (value-types);*
- *Ссылочные (referenced-types).*

К типам значений относятся все объекты структур, а к ссылочным — объекты классов. Отличие состоит в размещении объектов в памяти: объекты значимых типов размещаются в стеке целиком, тогда как переменная ссылочного типа сохраняется в стеке и хранит адрес объекта, который в действительности расположен в «управляемой куче» (области оперативной памяти, выделяемой для хранения относительно больших объёмов данных). При создании копии переменная значимого типа возвращает дубликат объекта, с которым связана, тогда как переменная ссылочного типа возвращает ссылку на объект. Операции с переменными значимого типа осуществляются быстрее чем операции с ссылочными типами. Исходя из этого, различаются и условия их применения, в случае, когда необходимы небольшой объем памяти и большее быстродействие используются значимые типы, если нужен большой объем динамической памяти, то используют ссылочные типы.

11. 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 `double` к типу `float` заключается в отсечении той части дробного числа, хранение которой не позволяет тип данных `float`. Конечно, возможна ситуация, при которой в переменной типа `double` хранится 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 приведён к типу
float, поскольку тип double более точный, то
процедура приведения типа происходит с потерей
информации, поэтому необходимо указывать суффикс
F после инициализирующего значения (о суффиксах
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 на рисунке 11.1 (*"Cannot implicitly convert type*

'float' to type 'int'.»; «Не могу неявно привести тип 'float' к типу 'int'»).

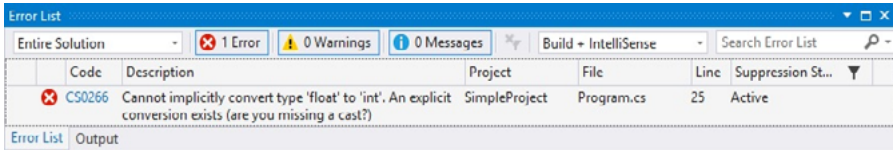


Рисунок 11.1. Ошибка компилятора

Для выполнения указанной операции без ошибки, необходимо, чтобы переменная "А" имела тип данных float, как на примере ниже:

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

Неявно допустимо приводить друг к другу следующие типы данных:

Из типа	К типу
<i>byte</i>	short, ushort, int, uint, long, ulong, float, double, decimal
<i>sbyte</i>	short, int, long, float, double, decimal
<i>short</i>	int, long, float, double, decimal
<i>ushort</i>	int, uint, long, ulong, float, double, decimal
<i>int</i>	long, float, double, decimal
<i>uint</i>	long, ulong, float, double, decimal
<i>long</i>	float, double, decimal
<i>ulong</i>	float, double, decimal
<i>char</i>	ushort, int, uint, long, ulong, float, double, decimal
<i>float</i>	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). Для 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.

Существует еще один способ осуществления преобразования типов. Это возможно благодаря наличию у всех базовых типов метода `Parse()`. Для осуществления преобразования необходимо передать в этот метод приводимое значение, пример использования приведен ниже:

```
//выводим пользователю сообщение
Console.Write("Введите целое число: ");
//получаем строку из консоли в строковую переменную
string numberString = Console.ReadLine();
//конвертируем строковое значение в числовое
int number = Int32.Parse(numberString);
//выводим результат
Console.WriteLine("Число = " + number);
```

Отличие между этими двумя способами следующее — методы класса `Convert` осуществляют преобразование не только на основании строковых значений, тогда как метод `Parse` работает только со строками.

12. Операторы

Операторы — это лексемы, которые на уровне языка программирования являются псевдонимами некоторых операций (как правило, «примитивных»).

Операторы необходимы для упрощения формулировки выражений включающих базовые операции. Например, математических выражений:

```
//объявление локальных переменных
int x = 5, y = 6, j = 7, z = 4;
//выполнение вычислений с использованием операторов
int result = (x + y) / j * z;
Console.WriteLine(result); // результат выполнения: 4
```

Из примера видно без дополнительных объяснений, что использование операторов значительно упрощает формулировку часто используемых выражений. Это относится не только к математическим выражениям. Например: такую часто используемую операцию, как конкатенация строк тоже можно условно выразить через символ "+" (операция сложения), которая интуитивно понятна в любом контексте. В языке программирования C# для типа данных string перегружен оператор +, выполняющий конкатенацию строк. Пример его использования показан ниже:

```
string FirstWord = "Hello";
string SecondWord = "world";
char Separator = ' ';
string ResultStatement = FirstWord + Separator +
    SecondWord;
```

```
Console.WriteLine(ResultStatement); // ВЫВОДИТ
//сообщение: "Hello world"
```

Ниже приведены категории операторов языка программирования C#:

Категории операторов	Операторы
Арифметические	+ - * / %
Логические (булевы и побитовые)	& ^ ! ~ && true false
Инкремент, декремент	++ --
Сдвиг	<< >>
Операторы отношения	== != < > <= >=
Операторы инициализации (присваивания)	= += -= *= /= %= &= = ^= <<= >>=
Доступ к компоненту класса (объекта)	.
Индексатор	[]
Ограничитель блока	()
Условный (тернарный) оператор	?:
Добавление и удаление делегата	+ -
Создание объекта	new
Получение информации о типе данных	as is sizeof typeof
Контроль ошибки переполнения	checked unchecked
Получение адреса и разыменование	* -> [] &

В настоящем уроке мы будем рассматривать *такие группы операторов* как:

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

Аргументы оператора называются «операндами». В качестве операндов выступают литералы указанные справа

и (или) слева от оператора. Выражение, которое допустимо указывать слева от оператора, называется *лево-допустимым* (*l-value*), а выражение, которое допустимо указывать справа от оператора — соответственно, *право-допустимым* (*r-value*). Например: слева от оператора присваивания обязательно должна находиться переменная, тогда как справа от него может находиться любое выражение, результатом которого будет значение того же типа, что и лево-допустимое выражение данного оператора.

```
int a = 5 / 3 + 4;
//a = "hello world"; — ошибка времени компиляции
//5 + 4 = a; — ошибка времени компиляции
Console.WriteLine("result is " + a); // выводит в
//консоль: result is 5
```

По количеству принимаемых операндов операторы делятся на:

- 1) унарные, то есть принимающие один операнд;
- 2) бинарные — два операнда;
- 3) тернарный — соответственно, три операнда.

Некоторые операторы, в зависимости от контекста употребления, могут выступать как в унарной так и в бинарной форме. Например, оператор "-" в бинарной форме выполняет роль оператора вычитания, а в унарной — переводит число в отрицательную форму, что показано на приведенном ниже примере:

```
int number = -3;
Console.WriteLine(number); // -3
number = 7 - 2;
Console.WriteLine(number); // 5
```

Контекст употребления оператора имеет большое значение. Например, выбор метода, определяющего действие оператора в каждом конкретном случае, зависит от типов данных его операндов. Количество и типы данных операндов описывают сигнатуру метода, который необходимо найти компилятору, чтобы выполнить операцию, указанную пользователем (программистом). В представленном ниже примере, в первом случае, происходит сложение двух чисел, поскольку в качестве операндов используются числа, а во втором — конкатенация строк, потому, что в качестве операндов использованы строки.

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

Арифметические операторы

Арифметические операторы используются для формулирования (описания) математических (арифметических) выражений и осуществления вычислений.

Арифметические операторы 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	

Действие бинарных арифметических операторов, а так же унарных «+» и «-», интуитивно понятно и ничем не отличается от действия аналогичных операторов в известном Вам языке программирования C++, поэтому мы не будем подробно на них останавливаться.

Отдельного внимания заслуживают операторы инкремента (++) и декремента (--) из-за их высокого приоритета. Действие названных операторов заключается в увеличении (инкременте) или уменьшении (декременте) операнда на некоторую заданную величину (по умолчанию на единицу; само собой разумеется, что величина зависит от метода, определяющего действие оператора). Приоритет является следствием того, что эти операторы могут употребляться в двух формах: префиксной (то есть перед операндом) и постфиксной (после операнда). Форма употребления влияет на последовательность обработки операторов всего выражения. На примере показано, что если используется префиксная форма оператора инкремента, то его обработка происходит до обработки всего выражения, в котором он используется (выражение №1), а если используется постфиксная форма, то оператор инкремента обрабатывается после того, как будет обработано выражение (выражение №2). Всё вышесказанное справедливо и для оператора декремента.

```
//объявление и инициализация локальных переменных
int x = 5, y = 6, result = 0;
//выражение №1
result = y - ++x;
Console.WriteLine("result = " + result); // result = 0
//восстановление начального значения переменной "x"
```

```
x = 5;
//выражение №2
result = y - x++;
Console.WriteLine("result = " + result); // result = 1
```

Операторы отношений

Операторы отношений — это операторы, которые используются для формулирования (выражения) неравенств; операторы, принимающие два сравнимых объекта и возвращающие логическое значение, которое указывает — верно ли неравенство.

Все операторы отношения являются бинарными по той причине, что существование отношения возможно только между двумя и более объектами. Но все множественные отношения принципиально сводимы к сложному выражению, состоящему из бинарных отношений, поэтому существование операторов описывающих множественные отношения является избыточным. Ниже, в таблице, приведены операторы отношений языка программирования C#.

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

Операторы отношений «==» и «!=» применимы ко всем типам данных, тогда как остальные операторы отношений могут применяться только к числовым типам данных.

```

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!!!

```

Необходимо помнить разницу между операторами отношений и логическими операторами. Отличие состоит в том, что аргументами операторов отношения являются любые сравнимые объекты, тогда как логические операторы принимают только логические величины (то есть значения типа «bool»: true или false).

Логические операторы

Логические операторы — операторы, используемые для формулирования выражений булевой алгебры; операторы, принимающие и возвращающие логическое значение; операторы, выполняющие логические операции, то есть действия, результатом которых порождаются новые понятия. К логическим операциям относятся: операция конъюнкции (логическое «умножение», логическое «И»), операция дизъюнкции (логическое «сложение», логическое «ИЛИ»), а так же логическое отрицание (логическое «НЕ»).

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

Ниже представлена таблица истинности для логических операторов:

&& и &	true && true = true;
	true && false = false;
	false && true = false;
	false && false = false;
и	true true = true;
	true false = true;
	false true = true;
	false false = false;
!	!true = false
	!false = true

Сокращённые логические операции (сокращённое «И», а так же сокращённое «ИЛИ») отличаются от полных тем, что второй операнд не обрабатывается, если это не необходимо. Например: при использовании логического «И» нет смысла выполнять обработку второго операнда, если первый операнд ложен, поскольку вне зависимости от того, какое значение принимает второй операнд — результатом выражения будет ЛОЖЬ (false). Если в выражении с логическим «ИЛИ» первый операнд — ИСТИНА, то и результат выражения будет ИСТИНА (true). Примеры приведены ниже:

```
bool FirstOperand = false, SecondOperand = false;
Console.WriteLine(FirstOperand && SecondOperand);
//False
SecondOperand = true;
Console.WriteLine(FirstOperand & SecondOperand);
//False
FirstOperand = true;
Console.WriteLine(FirstOperand & SecondOperand);
//True
```

На следующем примере демонстрируется использование логических операторов в выражениях различного вида:

```
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(false == true); // False
Console.WriteLine(2 > 7 && 2 / a != 5); // False
/* Нет ошибки! Второй операнд не вычисляется,
так как результатом в первом операнде уже является
false*/
/*Console.WriteLine(2>7 & 2/a!=5);
Ошибка: Попытка деления на ноль. Вычисляются оба
операнда, причем вычисление второго приводит
к ошибке*/
```

Битовые операторы

Для числовых типов в языке программирования C#, как и в большинстве других языков программирования, определены битовые или поразрядные операторы — операторы, которые поразрядно формируют результирующее число на основании соответствующих разрядов своих операндов. Они делятся на поразрядные булевы операторы и операторы побитового сдвига. Каждый бит операндов булевых операторов воспринимается как логическое значение: ПРАВДА (true = 1) или ЛОЖЬ (false = 0). Тогда как операторы сдвига, просто сдвигают разряд лево-допустимого выражения на значение, указанное соответствующим разрядом право-допустимого выражения.

В таблице приведены битовые операторы языка программирования C#:

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)

Ниже приведена таблица истинности для бинарных булевых операторов:

Таблица истинности для И (&)			Таблица истинности для включающего ИЛИ ()			Таблица истинности для исключающего ИЛИ (^)		
1 операнд	2 операнд	Результирующий бит	1 операнд	2 операнд	Результирующий бит	1 операнд	2 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

На приведённом ниже примере демонстрируется использование битовых операторов языка программирования C#:

```
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

Назначение оператора присваивания состоит в том, чтобы присвоить лево-допустимому выражению значение право-допустимого.

В языке программирования C# определены, так называемые, составные операторы присваивания (их ещё называют вычислением по сокращённой схеме). Эти операторы совмещают в себе действие оператора присваивания и любой другой бинарной операции, допустимой для числовых типов данных.

Например, объявлены переменные:

```

int a = 10;
int result = 5;

```

тогда операция:

```

result += b;

```

будет сочетать в себе сложение переменной «result» и переменной «a», с последующим присвоением результата сложения переменной «result». Иными словами, лево-допустимое выражение «result» играет роль лево-допустимого выражения и в операции сложения и в операции присваивания. Визуально описанную операцию можно преобразовать в выражение:

```
result = result + b;
```

Ниже приведены доступные в языке C# составные операторы присваивания:

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

Приведённый ниже пример иллюстрирует использование операторов присваивания в языке программирования C#:

```
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 выражении, при условии, что приоритет не указан явно (при помощи оператора «()»). В приведённой ниже таблице приоритет операторов изменяется от высшего к низшему:

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

Приведённый ниже пример иллюстрирует влияние приоритета операторов на последовательность их обработки в арифметических выражениях:

```
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); //19
result = (a + (b - 4)) * -2;
Console.WriteLine(result); //-14
```

13. Условия

Условие — это логическое выражение (выражение, результатом которого является логическое значение), которое используется для реализации алгоритма ветвления. Условие — величина логическая, в том смысле, что не выражается в виде некоторой лексической единицы. Ветвление имеет место в тех случаях, когда необходимо принять решение о выборе альтернативного действия (то есть одного из взаимоисключающих действий) либо о том, выполнять некоторое действие или не выполнять.

Условия (а так же условные конструкции, которые на них основаны), как средство управления выполнением программного кода, используются на всех уровнях абстракции.

Простое условное выражение состоит из одного неравенства, тогда как составное условие является последовательностью логических выражений.

```
a > b // простое условие  
a > b && b > c || x == y // составное условие
```

Условный оператор if

Условные операторы (операторы ветвления) предназначены для реализации ветвления программного кода, то есть для создания альтернативных программных ветвей (взаимоисключающих сценариев, действий, процедур), описывающих различное поведение алгоритма в различных условиях.

Конструкция ветвления «if» имеет вид:

```
if      (Условное Выражение)
{
    Действие;
}
```

Необходимо обратить внимание на то, что результатом условного выражения обязательно должно быть значение типа «bool», поскольку в языке C#, в отличие от языка C++, тип «bool» не сводим к целочисленному типу данных. Поэтому компилятор проверяет, чтобы в качестве условного выражения использовались только логические значения. Следующий ниже пример показывает, что использовании в качестве условного выражения типа данных, отличного от типа «bool», возвращается ошибка на этапе компиляции:

```
class Program
{
    static void Main(string[] args)
    {
        int f = 1;
        if (f) // Ошибка: попытка перевести int в bool.
        {
            Console.WriteLine(f);
        }
    }
}
```

В приведённом примере конструкция «if» используется для проверки не равно ли значение переменной «f» нулю перед тем, как вывести в консоль её значение, как это принято в C++. И, естественно, такая формулировка является ошибочной в C#. Для того, чтобы проверить

не равна ли переменная нулю, необходимо сравнить её значение со значением 0 как это показано на нижеследующем примере.

```
class Program
{
    static void Main(string[] args)
    {
        int f = 1;
        if (f != 0)
        {
            Console.WriteLine(f); // 1
        }
    }
}
```



Блок-схема хорошо иллюстрирует, что конструкция «if» создаёт одну отдельную ветку программного кода, вход в которую осуществляется в том случае, если условное выражение возвращает «true».

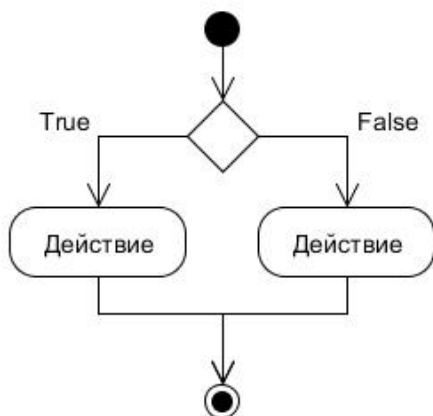
Ниже приведён пример, который иллюстрирует использование конструкции «if» на практике (поздравле-

ние выводиться только в том случае, если игрок угадал значение сгенерированное программой).

```
int MAX = 10;
Console.Write("Угадайте число от 1 до {0}...",
MAX);
int userNumber = Convert.ToInt32(Console.
ReadLine());
Random rnd = new Random();
double PcNumber = rnd.NextDouble() * MAX;
PcNumber = Math.Round(PcNumber);
Console.Write("Правильное число {0}, а вы задали
{1}.\n", PcNumber, userNumber);
if (PcNumber == userNumber) // Число угадано!
{
    Console.WriteLine("Поздравляем!");
}
```

Условный оператор if else

Конструкция ветвления «if else», в отличие от конструкции «if», разделяет общий поток выполнения на две альтернативные ветви кода. Иными словами,



если условие выполняется, то есть условное выражение возвращает «true», то выполняется блок «if», в противном случае выполняется альтернативная ветка кода (блок «else»).

Конструкция «if else» имеет вид:

```
if (Условное Выражение)
{
    //Действия 1;
}
else
{
    //Действия 2;
}
```

На следующем примере иллюстрируется использование конструкции «if else»:

```
class Program
{
    static void Main(string[] args)
    {
        int f = 1;
        if (f > 0)
        {
            Console.WriteLine(f);
        }
        else
        {
            Console.WriteLine("Test");
        }
    }
}
```

Условный оператор switch

Если конструкция «if» в качестве условного выражения принимает логическое значение, то конструкция «switch» выполняет сравнение переданного аргумента (в роли аргумента может выступать любая скалярная переменная или константа) с некоторыми константными величинами, называемыми случаями (cases), и если значение переменной совпадает с одним из случаев, то выполняется соответствующий блок кода (case-блок). В случае, когда ни одно «case-значение» не совпало со значением аргумента, выполняется блок «default».

Конструкция «switch» имеет следующий вид:

```
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;
}
```

Каждый «case-блок» должен закрываться либо инструкцией «break», прекращающей обработку конструкции «switch», либо инструкцией «return», прекращающей выполнение метода, в котором обрабатывается текущая

конструкция «switch». Блок «case» может быть не закрытым только в том случае, если он не имеет тела (в нём не указаны никаких действий; так же говорят что такой «case-блок» пуст). В таком случае осуществляется переход к ближайшему последующему непустому «case-блоку», как показано на приведённом ниже примере. Если непустой блок не «закрывается», то возвращается ошибка на этапе компиляции:

```
int num = 1, price;
switch (num)
{
    //Если количество товара до 4 штук,
    //то цена 25 Коп за 1 единицу товара.
    case 1:
        /*если разкомментировать следующую строчку,
        то возвращается ошибка на этапе компиляции*/
        //price = 10;
    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;
}
```

Console.WriteLine(price); // 25

Тернарный оператор ?:

Единственный в С# тернарный оператор «?:» применяется следующим образом:

```
Условие ? выражение№1 : выражение№2
```

В качестве первого операнда принимает логическое выражение (которое играет роль условного выражения) и, если результатом условного выражения является «true», то оператор «?:» возвращает выражение №1, в противном случае возвращается выражение №2.

Ниже приведён пример использования условного оператора «?:».

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

14. Циклы

Циклические конструкции, или просто циклы, языка программирования используются для организации повторяющихся действий.

Существует два подхода к организации повторяющихся процедур, другими словами — зацикливания: итерационный и рекурсивный.

Смысл итерационного подхода состоит в том, что некоторая процедура (итерация) заключается в условный блок и выполняется до тех пор, пока выполняется условие.

Рекурсивный метод заключается в условном самовывозе процедуры (то есть, процедура вызывает сама себя, при выполнении определённого условия) и реализуется посредством «рекурсивных методов». Поскольку, не существует языковых механизмов реализации самовывоза части процедуры, очевидно, что речь в данной главе пойдёт об итерационных конструкциях, которые, для упрощения терминологии, называют попросту «циклами».

Одноразовое выполнение тела цикла называют «итерацией» (*iteration*).

Переменную, которая определяет количество итераций, называют «итератором» (*iterator*) или «счётчиком цикла» (*cycle counter*).

Условие, определяющее выход из циклической конструкции, называют «условием выхода» (*exit condition*). В зависимости от того, когда выполняется проверка условия выхода, циклы делят на:

- циклы с предусловием (условие проверяется до того, как будет выполнено тело цикла);
- циклы с постусловием (условие проверяется после того, как выполняется тело цикла).

По наличию в цикле итератора, выделяют:

- Циклы со счётчиком;
- Циклы без счётчика.

По наличию условия выхода выделяют:

- Условные циклы;
- Безусловные или бесконечные циклы.

Так же выделяют отдельную группу циклов, которые называют «циклами коллекции» или «циклами по коллекции» (так же их называют «циклами просмотра» или «циклами прохода» *от английского «walkthrough cycles»*) — это циклические конструкции, которые предназначены для выполнения определённой процедуры над всеми элементами некоторого множества (обычно множество представлено некоторым массивом данных). Представителем этой группы циклических конструкций в С# является цикл «foreach» («for each» — *с англ. «для каждого»*).

Для бесконечных циклов не создано специальной конструкции, которая бы их реализовывала, поскольку бесконечный цикл можно организовать на базе любой циклической конструкции, обеспечив ей условное выражение, которое будет выполняться во всех случаях (всегда возвращать значение «true»). Пример бесконечного цикла будет показан дальше в разделе, посвящённом циклу «while», поскольку циклическая конструкция «while»

позволяет организовать бесконечное заикливание самым простым и «прозрачным» способом.

В языке программирования C# определены следующие итерационные конструкции: «for», «while», «do while» и «foreach»; инструкция «goto» относится к инструкциям перехода, однако, с её помощью можно организовать условное заикливание части процедуры.

Цикл for

Цикл «for» относится к группе циклов с предусловием и является циклом со счётчиком. Цикл «for» в общем виде выглядит следующим образом:

```
for (инициализация_переменных; условное_выражение;  
выражение) // Заголовок цикла;  
  
{  
    // Тело цикла;  
}
```

Заголовок цикла содержит три обязательных блока: блок инициализации локальных переменных, блок, содержащий условие выхода, и блок, содержащий выражение определяющее шаг цикла.

Как правило, в заголовке цикла «for» фигурирует одна переменная, которая является счётчиком цикла (итератором), как показано на приведённом ниже примере:

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

Однако бывают ситуации, в которых необходимо объявить более чем одну локальную переменную цикла, как показано на примере:

```
for (int x = 0, y = 0; x < 10 && y < 10; x++, y++)
{
    /*Ваш код здесь*/
}
```

В заголовке цикла следует объявлять только те переменные, которые итеративно изменяются в цикле. Если переменная не связана с итерациями цикла, то объявлять её в заголовке цикла избыточно и считается дурным тоном, поскольку затрудняет чтение и понимание кода.

Необходимо помнить, что в качестве условного выражения, как и во всех условных конструкциях языка C#, обязательно должно использоваться логическое значение (логическое выражение).

Ниже представлен пример алгоритма, использующего цикл «for». Назначение алгоритма состоит в том, чтобы определить, является ли введённое слово палиндромом.

```
int counter = 0;
string str = "кабак";
//Сравниваем первую букву с последней,
//вторую с предпоследней и т.д.
for (int i = 0, j = str.Length - 1; i < j; i++, j--)
{
    if (str[i] == str[j])
        counter++;
    //если буквы равны, прирачиваем счетчик
}
```

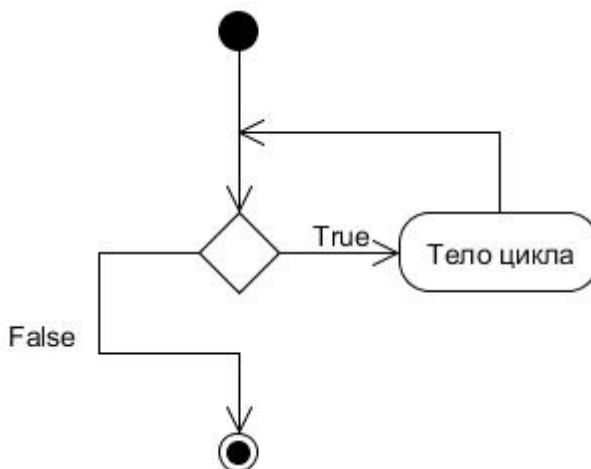


```
//если счетчик равен длине_строки/2,
//то строка является палиндромом
if (counter == str.Length / 2)
{
    Console.WriteLine("Строка палиндром");
}
else
{
    Console.WriteLine("Строка не палиндром");
}
```

Цикл while

Цикл «while» относится к группе циклов с предусловием и является циклом без счётчика. Он «while» имеет следующий вид:

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



На блок-схеме видно, что тело цикла будет выполняться пока и только в том случае, если выполняется условное выражение.

Следующий ниже пример показывает, как на базе циклической конструкции «while» организовать бесконечный цикл.

Данный алгоритм будет бесконечно выводить в консоль случайные числа, пока приложение не будет принудительно завершено:

```
Random rand = new Random();
while (true)
{
    Console.WriteLine(rand.Next());
}
```

Принято считать, что цикл «while» относится к циклам без счётчика. Однако можно ввести в него псевдо счётчик, как показано на примере ниже:

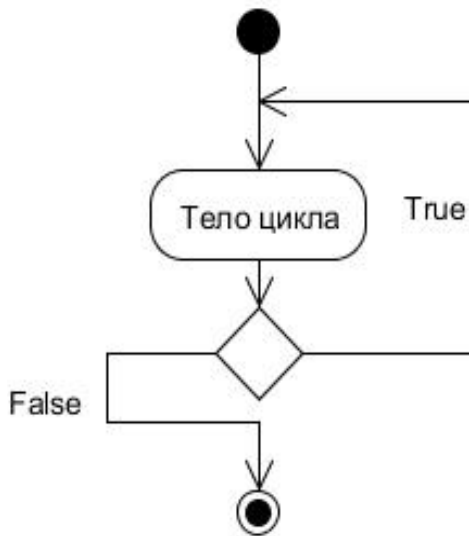
```
// Сложение чисел от 0 до 20 пока сумма не будет
// равна 100
int counter = 0; //счетчик для чисел
Random rand = new Random();
int number; //переменная для хранения числа
int summ = 0; //переменная для хранения суммы
while (summ <= 100) //пока сумма меньше 100
{
    number = rand.Next(0, 20); //генерируем число
    summ += number; //добавляем к сумме
    counter++; //прибавляем счетчик
}
Console.WriteLine(
    "Сумма {0} чисел от 0 до 20 равна {1}.", counter, summ);
```

Цикл do while

Конструкция «do while» относится к циклам с постусловием и имеет вид:

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

Блок-схема иллюстрирует, что тело цикла «do while» находится в потоке исполнения выше условного выражения, а значит, в отличие от циклической конструкции «while», его тело будет выполнено как минимум один раз, даже если условное выражение «всегда ложно».



Ниже представлен пример, иллюстрирующий применение конструкции «do while» на практике:

```
// посчитать сумму всех цифр любого числа
int summ = 0; //переменная для суммы
Console.WriteLine("Введите любое целое число:");
int number = Convert.ToInt32(Console.ReadLine());
//любое число
int temp = number; //Запоминаем число
do
{
    summ += number % 10;
    //находим последнюю цифру и суммируем её
    number /= 10;
    //отсекаем последнюю цифру от числа
} while (number > 0);
//если number больше нуля, вернуться
//и повторить операторы тела цикла
Console.WriteLine("Сумма всех цифр числа {0} = {1}",
temp, summ);
```

Цикл foreach

Как уже было сказано выше, цикл «foreach» представляет группу циклов «по коллекции» и предназначен для выполнения некоторой процедуры для всех элементов определённого массива. Циклическая конструкция «foreach» описывается следующим образом:

```
foreach (тип_переменной идентификатор in контейнер)
{
    Действие;
}
```

В заголовке цикла объявляется переменная, которая, разумеется, должна иметь тот же тип данных, что и массив-контейнер. Из контекста ясно, что «контейнером» всегда должен быть некоторый массив, ссылка на

который указывается после частицы «in», выполняющей в данном выражении роль союза, который отделяет объявление переменной, от указания используемой циклом коллекции.

В каждой отдельной итерации цикла в объявленную локальную переменную помещается ссылка на определённый элемент массива, в случае перебора массива ссылочного типа, или дубликат элемента — в случае обработки массива значащего типа.

Ниже приведён пример использования циклической конструкции «foreach» в языке программирования C#:

```
/// 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);
```

Для расширения возможностей циклических конструкций и внесения большей гибкости в описание циклов, в языке C#, как и во многих других языках программирования (например C++), предусмотрены две инструкции выхода, которые позволяют досрочно прервать как выполнение всего цикла, так и выполнение текущей итерации: «break» и «continue».

Инструкция break

Инструкция «break» используется для прерывания выполнения всего цикла.

В приведённом ниже примере инструкция «break» используется для того, чтобы на базе бесконечного цикла организовать вывод чисел кратных «6» в интервале от «0» до «100», с последующим выходом из цикла:

```
int i = 0;
while (true)
{
    if (++i % 6 == 0)
    {
        Console.WriteLine(i);
    }
    if (i == 100)
    {
        break;
    }
}
```

Инструкция continue

Инструкция «continue» используется для прекращения выполнения текущей итерации цикла и перехода к следующей. В представленном ниже примере инструкция

«continue» используется для вычисления всех четных чисел в промежутке от 1 до 20.

```
for (int i = 1; i < 20; i++)
{
    if (i % 2 != 0)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

В следующем примере инструкции «break» и «continue» используются совместно в одном цикле: «break» используется для прекращения выполнения цикла (выхода из блока получения данных у пользователя) и перехода к подсчёту среднего значения, а инструкция «continue» — для перехода к следующей итерации в случае, если введённое значение меньше необходимого минимума.

```
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("Введите зарплату
        больше чем 500 грн.\n (для окончания
        введите отрицательное значение):\n");
    zarplata=Convert.ToDouble(Console.ReadLine());
    if(zarplata < 0)
        break;
    if(zarplata < 500)
        continue;
    else
    {
```

```

        sr_zarplata += zarplata;
        kol++;
    }
}
while(true);
if(kol > 0)
{
    sr_zarplata /= kol;
}
Console.WriteLine("Средняя зарплата = {0}",
                    sr_zarplata);

```

Инструкция goto

Инструкция перехода «goto» используется для реализации «условного отката» по процедуре. Инструкция «goto» состоит из метки и осуществления перехода к выбранной метке. В качестве метки используется некоторый произвольный идентификатор:

```

Label1: // метка, к которой будет осуществлён переход
... /* операции, выполняемые между меткой
и переходом к метке */
if (условное_выражение) // условие перехода
{
    goto Label1; // переход к метке
}

```

Необходимо помнить, что перед тем, как использовать идентификатор, его необходимо объявить. Таким образом, объявление метки должно обязательно предшествовать переходу к ней. Данное замечание может показаться избыточным, в силу его очевидности. Но, как ни странно, переход к метке, которая объявлена ниже

в потоке выполнения, а так же переход с нарушением области видимости являются распространёнными ошибками.

Как правило, переходу к метке предшествует некоторая условная конструкция, поскольку, если переход выполняется безусловно, то инструкция «goto» организует бесконечный цикл.

Несмотря на универсальность инструкции «goto», её использование считается «плохим тоном программирования», поскольку избыточное применение «goto» путает код, и он становится сложен для восприятия. Поэтому необходимо помнить о «чувстве меры» и целесообразности применения тех или иных методов и приёмов.

Наиболее часто «goto» используется «внутри» конструкции «switch», для перехода между «case-блоками», поскольку формально «case-блок» является меткой, что и демонстрируется на следующем примере.

```
int level = Int32.Parse(Console.ReadLine());
switch (level)
{
    case 0:
        Console.WriteLine("Уровень 0");
        break;
    case 1:
        goto case 2;
    case 2:
        Console.WriteLine("Уровень от 1 до 2");
        goto default;
    default:
        Console.WriteLine("Пока");
        break;
}
```

Домашнее задание

1. Даны целые положительные числа A , B , C . Значение этих чисел программа должна запрашивать у пользователя. На прямоугольнике размера $A \times B$ 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 10000 грн. 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 11000 грн., 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 большее 0, найти число, полученное при прочтении числа N справа налево. Например, если было введено число 345, то программа должна вывести число 543.