# Оглавление

[**Оглавление**](#_8y22pz1dd6rt) **1**

[**Глава 1 Основы Computer science**](#_tr52lfmkt931) **6**

[**Глава 2 Курс программирования на C# Основы программирования (C Sharp)**](#_zewq87s6ul0p) **6**

[**2. Система типов языка С#**](#_4rnua9up0ig3) **6**

[2.1 Примитивные типы в языках программирования](#_tff3wq233ayq) 6

[2.1.1 Примитивный тип данных dynamic](#_7jgbbmlhgt7z) 8

[2.2 Преобразование и приведение типов](#_u0560c9z25x3) 10

[2.3. Ключевые слова, идентификаторы, литералы](#_z9hdqaj3kziu) 11

[2.3.1 Ключевые слова checked и unchecked](#_vnecgzajytof) 11

[2.3.1.1 Про тип Decimal и BigInteger](#_yorsy3f0e3yq) 12

[2.3.2 Ключевое слово var](#_xaqf8bhzitfj) 12

[2.3.3 Ключевое слово static(нет ответа)](#_jeg8w9szip9t) 13

[2.3.4 Ключевые слова C#, относящиеся к управлению версиями](#_ourmdhgntunm) 13

[2.3.6 Ключевое слово implicit](#_ifywrc7zqpjt) 13

[2.3.7 Ключевое слово explicit](#_mslr2cu5sxs6) 13

[2.3.8 Ключевое слово default (нет ответа)](#_mvyvgwm6zz3w) 14

[2.3.9 Ключевое слово ref или out (нет ответа)](#_1w1ymumlurfl) 14

[2.3.10 Литералы](#_ukjavc2ebcwd) 14

[2.4 Ссылочные и значимые типы](#_l9xy7upwq420) 14

[2.4.1 Члены типа](#_ws87m211m1i3) 18

[2.4.2 Видимость типа](#_7jkdf3h7mvg6) 21

[2.4.3 Дружественные сборки (не понял)](#_w1x3cou6k0rt) 21

[2.5 Выражения и операции(нет ответа)](#_d060a7hjc0ej) 21

[2.6 Константы](#_c9gsn62qtpx) 21

[2.7 Поля](#_cshpxkt8q9oy) 22

[2.7 Переменные (нет ответа)](#_sn4nzzw111j3) 25

[**3. Модификаторы доступа**](#_9qlsoqyj7ql) **25**

[3.1 Доступ к членам типов](#_71qzfmxfl2an) 25

[3.2 Модификаторы доступа полей](#_4jrz761xl79b) 26

[**4. Классы**](#_577f5hkdbyeh) **26**

[4.1 Статические классы](#_v79ppruooyrm) 26

[4.2 Частичные классы](#_coc796cola35) 27

[4.3 Запечатанные классы](#_97pvxuper9o8) 28

[**5 Методы**](#_pq5orzdcsf1) **30**

[5.1 Вызов виртуальных методов, свойств и событий в CLR](#_j19tcrj6zth) 30

[5.2 Работа с виртуальными методами при управлении версиями типов](#_zbnsy38r6d41) 33

[5.3 Конструкторы экземпляров и классы (ссылочные типы)](#_bo8as8imm8sw) 37

[5.3.1 Конструктор по умолчанию (без параметров)](#_p7zkjcx4fy4t) 37

[5.3.2 Создание конструктор для ссылочных типов (class)](#_1s5bihlz2xif) 38

[5.3.3 Конструкторы значимых типов (struct)](#_sni7vqrvz476) 40

[5.3.4 Конструкторы типов (статические конструкторы, конструкторы классов и инициализаторы типов)](#_qle4zj8xxjvo) 43

[5.4 Методы перегруженных операторов](#_5szl6lq6065z) 45

[5.5 Методы операторов преобразования](#_5zm2s65cp3gj) 47

[5.6 Методы расширения](#_h1he2rlyl8i6) 50

[5.6.1 Правила и рекомендации](#_foybpu2r1889) 52

[5.6.2 Расширение разных типов методами расширения](#_9z1cau52v9qh) 53

[5.6.3 Атрибут расширения](#_lncg3of6ffdq) 55

[5.7 Частичные методы](#_werwvq202rsd) 56

[5.7.1 Правила и рекомендации](#_ej0pxzh9s5se) 58

[**6 Параметры**](#_h2cty8fhbp8c) **59**

[6.1 Необязательные и именованные параметры (аргументы)](#_er42sm6kru5c) 59

[6.1.1 Правила использования параметров](#_p0n3lvagv8sg) 60

[8.9. Класс System.Object](#_sqrfqga9c10b) 63

[8.9.1 Равенство и тождество объектов Equals](#_ntkkx6chfvvw) 63

[8.9.2 Хеш-коды объектов GetHashCode](#_3aq3ez4w5t1u) 65

[**Глава 3 Курс программирования на C# Основы ООП (C Sharp)**](#_xeqto0puh93c) **67**

[**1. Основы ООП**](#_bw0i1cvzyayd) **67**

[1.3. Пространство имен](#_lx2039772udy) 67

[9.3. Операции упаковки, распаковки](#_7ir2z2w98t5d) 71

[9.3.1 Упаковка значимых типов](#_fl361om34wof) 71

[9.3.2 Распаковка значимых типов](#_f73oa2yy8y36) 72

[**Глава 4 Промышленное программирование на ASP.NET (курс EPAM)**](#_zdqi92gtup6s) **79**

[**1. Инструментарий разработчика (нет ответа)**](#_adrwpp46f3gq) **79**

[1.1 Visual Studio 2017 (нет ответа)](#_4kct5uc1csp2) 79

[**2. Введение в Asp.NET MVC 5**](#_il6otsp4p36y) **79**

[1.1 Коротко о браузере](#_ukxohnc9qgl) 79

[1.2 Знакомство с работой HTTP протокола](#_ygujfa5lgyye) 80

[1.2.1 Кратко о методе HTTP запроса](#_ra2le723qst7) 80

[1.2.2 Заголовок (HTTP заголовки)](#_vhi6014dqf2n) 80

[1.2.3 Тело](#_ooibt273tk52) 81

[1.2.4 Пример ответа](#_1paq4mssjmlh) 82

[1.2.5 Типы запроса (Методы запросов)](#_b6095rw7tazc) 82

[1.2.5 HTTPS](#_ftey7j31pemz) 82

[1.2.6 Кратко о HTTP2 (H2)](#_z6lfo6ubcfgc) 84

[1.2.7 Актуальные проблемы работы с HTTP (нет ответа)](#_oipbsxuqcgqp) 84

[1.2.8 Простейшие инструменты для анализа HTTP запроса (нет ответа)](#_eeu3n8bkfmcy) 84

[1.3. Нововведение в ASP .NET MVC 5](#_ydgksjrum7xl) 84

[1.4. Предназначение ASP .NET MVC 5](#_uc0gsoz3ica7) 85

[1.5 Обзор Asp.Net](#_2u33pegdrar3) 85

[1.5.1 Web Pages Applications. Веб-страницы ASP.NET](#_ze5fuqtonkqk) 85

[1.5.2 Web Forms Applications. Веб-формы](#_guat3rhn0vhf) 85

[1.5.3 MVC Applications.](#_j88a7g4uddne) 85

[1.5.4 Преимущества MVC приложения](#_fw8vf8jjr4o2) 85

[1.5.4.1 Расширяемость](#_xgg8uqju1nt0) 85

[1.5.4.2 Жесткий контроль над HTML и HTTP](#_unlyu8ksss8r) 86

[1.5.4.3 Тестируемость](#_li4ubfqbwgba) 86

[1.5.4.4 Мощная система маршрутизации (нет ответа)](#_d9bpa1huo54o) 86

[1.6 Общие возможности для Asp.Net приложений (нет ответа)](#_s0lcwbujrbzn) 86

[1.6.1 конфигурация (нет ответа)](#_l64h958ien1e) 86

[1.6.2 аутентификация (нет ответа)](#_xno1wg3wqdig) 86

[1.6.3 роли и провайдеры (нет ответа)](#_5ayl13p6ipsz) 86

[1.6.4 управление состоянием (нет ответа)](#_jck9wnll3cij) 86

[1.6.5 кэширование (нет ответа)](#_ctxpwzfwfbl4) 86

[1.7 Введение в MVC приложение](#_vqlob9j9ml36) 86

[1.7.1 Модель, представление, контроллер. Понимание паттерна MVC.](#_lclx9n5ifcez) 87

[1.7.1.1 Модель (Model)](#_2nivuxvd0b8s) 87

[1.7.1.2 Представление (View)](#_1o9u2wf7i7lp) 87

[1.7.1.3 Контроллер (Controller)](#_8n9ovtnsvle7) 88

[1.7.1.4 Понимание паттерна MVC.](#_eq3seuv7ualf) 88

[1.7.2 Различия MVC & MVVM & MVP](#_kv22735m3rs0) 89

[1.7.2.1 Model-View-Presenter](#_48f9qtpa35zv) 89

[1.7.2.2 Model-View-View Model](#_dshybdsnlre4) 90

[1.7.2.3 Model-View-Controller](#_hlw93bw3hvt2) 91

[1.7.3 Общие правила выбора паттерна](#_bqo37icmi93l) 92

[1.7.3.1 MVVM](#_wmkwtzashsaa) 92

[1.7.3.2 MVP](#_1001vyd4sdju) 92

[1.7.3.3 MVC](#_u1gmzqfc34t) 92

[1.7.4 Обзор новых возможностей в MVC разработке.( нет ответа)](#_2b7wc7a920l6) 92

[1.8 Хостинг Web приложений. Знакомство с IIS и IIS Express.](#_watdmgv29wbt) 92

[1.8.1 Как работает хостинг сайтов?](#_6qqkztqgwyv) 92

[1.8.2 Типы хостинга](#_5m8cak1q5tkt) 93

[1.8.2.1 Общий хостинг (Shared Hosting)](#_88q7ythkxuq6) 93

[1.8.2.2 VPS хостинг (Virtual Private Server – виртуальный приватный сервер)](#_d6nu20i3zx9a) 93

[1.8.2.3 Облачный хостинг (Cloud Hosting)](#_491blhqdlxb6) 93

[1.8.2.4 WordPress хостинг](#_83zoeijlcpn0) 94

[1.8.2.5 Выделенный хостинг](#_6lei3pd8q2ow) 94

[1.8.2.6 Колокация «Colocation»](#_i1w4h6t5ybhi) 94

[1.8.3 Знакомство с IIS, IIS Express и ASP.NET Development Server](#_y3hv9tx86c5m) 95

[1.8.3.1 ASP.NET Development Server](#_q5lcvku07akp) 95

[1.8.3.2 IIS веб-сервер](#_1j3ewz96o78c) 95

[1.9 Жизненный цикл запроса к серверу.](#_2fhwbj5b72gj) 96

[1.9.1 Работа конвейера веб-сервера IIS](#_viyckgsqvzft) 96

[1.9.1.1 Обработка запросов в IIS](#_g4nl3yhmwisb) 97

[1.9.2 Основы архитектуры IIS, или запросопровод для ASP.NET](#_y8txu6y9hsmx) 98

[1.9.2.1 Общий план](#_4icl66mnnwh5) 98

[1.9.2.2 Крупный план](#_hsww2k4nl8ng) 99

[1.9.3 ASP.NET MVC жизненный цикл запроса](#_r4dimzgsy69q) 105

[1.9.3.1 MVC цикл](#_8z6oaqvafjnn) 107

[1.9.3.2 Routing (Маршрутизация)](#_s5he0ldmusf0) 107

[1.9.3.3 Controller initialization (Инициализация контроллера)](#_poeoo3eq9uox) 109

[1.9.3.4 Action execution (Выполнение действия)](#_d34nafdwxttx) 112

[1.9.3.5 Result execution (Выполнение результата действия)](#_w4gnhvftco3f) 115

[1.9.4 Жизненный цикл приложения ASP.NET MVC](#_ncys3a4okci8) 118

[1.9.5 Дополнительная инфа](#_7bifqpok8imd) 120

[1.9.5.1 ASP.NET Advanced. Урок 1. Архитектура IIS. Жизненный цикл HttpApplication.](#_8i1o4qb9zq2j) 120

[1.9.5.2 Набором статей по этой теме](#_5oddire8fejq) 120

[**3. Создание MVC приложения**](#_x68y4v6k35gs) **120**

[3.1 Шаблоны MVC приложения(Project Template)](#_tuvq6njap3we) 120

[3.2 Структура проекта MVC 5 MVC в (Solution explorer)](#_vodvw0n9tuy2) 121

[3.3 Соглашения именования](#_aktldfe0b01z) 121

[3.3.1 Соглашение первого вида](#_p2lmol57vbq4) 121

[3.3.2 Соглашение второго вида](#_5phjzln7xnni) 122

[**4. Разработка моделей Asp.NET MVC**](#_518bwuam3ixf) **123**

[4.1 Понятие модели](#_sr181f3omhn4) 123

[4.2 Создание модели (нет ответа)](#_8chx248nm6kb) 123

[**5. Разработка контроллеров Asp.NET MVC**](#_8zkcnv45jtvj) **123**

[5.1 Взаимодействие контроллера с пользовательским запросом.](#_8c0h7nmfxan5) 123

[5.2 Маршрутизация запроса](#_p020fo1me1kn) 124

[5.2.1 Определение действий контроллера.](#_nm7akzi6j3su) 124

[5.2.2 Виды возвращаемого результата.](#_v2envhism2aw) 125

[5.2.3 Использование параметров в действиях](#_cr62jf31f4ix) 125

[4.2.3.1 Автоматическое связывание параметров методов](#_l19297751zrc) 126

[5.2.4 Передача информации в представление](#_5bzwlz8b8qx9) 126

[5.2.4.1 Генерация представлений Модель](#_qbhmypolc761) 127

[5.2.4.2 ViewBag (нет ответа)](#_ij451cv7gb20) 127

[5.2.4.3 TempData(нет ответа)](#_lcyl4ehjbqaw) 127

[**6. Разработка представлений Asp.NET MVC(надо отредактировать на что-то подобное)**](#_uw8gmi57zafe) **128**

[6.1 Создание представлений с помощью движка Razor](#_udcc5nk0cd6l) 128

[6.1.1 Создание нового представления.](#_ios03ygrwavc) 128

[6.1.2 Знакомство с Razor синтаксисом.](#_1vlzsupwsn9s) 128

[6.1.3 Использование HTML Helpers (нет ответа)](#_og7fn9uxiyqa) 129

[6.1.4 Шаблоны элементов представления. Повторное использование кода в представлениях (Мастер-страница)](#_xcu0tqbslm8f) 129

[6.1.4.1 Создание и использование шаблонов.](#_99gaij3odbyf) 129

[6.1.4.2 Использование ViewStart для отображения частичного представления](#_6sghsiawlhz6) 130

[6.1.5 Использование RenderAction()](#_s26l45w1vb0c) 131

[6.1.5.1 RenderBody()](#_ryun3i7xr6ln) 131

[6.1.5.2 RenderSection()](#_dekewohf0t9p) 131

[6.1.6 Слои и темы(Bundles)](#_lgazk4r3t24m) 132

[6.1.7 Частичное представление(Partial Views)](#_y3mfvv51efev) 133

[**7. Маршрутизация**](#_wnaro5ktj5qd) **133**

[7.1 Введение в маршрутизацию MVC (Routing)](#_bo6mu9rg0fe9) 133

[7.2 Введение в шаблоны URL](#_2qqghkiojwz7) 133

[7.2.1 Переменные сегментов](#_eegv5qv2fwh9) 134

[7.2.2 Ключевое поведение шаблонов](#_asd8ij2v1gik) 134

[**25 Bootstrap 3 (нет ответа)**](#_vrkipgmplyhs) **135**

[**28 ORM. ENTITY FRAMEWORK 6 (нет ответа)**](#_yas3avq9xocy) **136**

[**29 Asp.Net Identity Framework (нет ответа)**](#_jo6franjtm1x) **137**

[**30 WEB API 2 (нет ответа)**](#_ijxmrxmswcpw) **137**

[**31 OWIN И KATANA (нет ответа)**](#_lvt2rdn46urb) **137**

[**31 Библиотека SignalR. Push-уведомления (нет ответа)**](#_k5oebju28k0d) **138**

[**Глава 5 IT-Academy**](#_jsicxptaxgad) **139**

[**1. Первое занятие - 25.09.2019**](#_ibvry1r1lqqa) **139**

[1.1 Начало работы с BitBucket](#_y2q3jhvm9ov3) 139

[1.2 Начало работы с VisualStudio.](#_gqga55p0cz9x) 139

[1.3 Литература:](#_mrtyl8jbdg97) 139

[**2. Второе занятие - даты нет**](#_3yb3n4resp8w) **139**

# Глава 1 Основы Computer science

# Глава 2 Курс программирования на C# Основы программирования (C Sharp)

1. Основы программирования

1.1 Алгоритмы

1.2 Эволюция языков

1.3. Обзор платформы .NET

1.4. Язык C#, общие концепции синтаксиса

# 2. Система типов языка С#

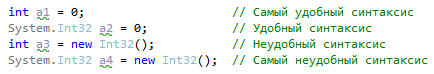
В этой главе речь идет о разновидностях типов, с которыми вы будете иметь дело при программировании для платформы Microsoft .NET Framework. Важно, чтобы все разработчики четко осознавали разницу в поведении типов. Приступая к изучению .NET Framework, я толком не понимал, в чем разница между примитивными, ссылочными и значимыми типами, в результате мой код получался не слишком эффективным и содержал много коварных ошибок. Надеюсь, мой опыт и мои объяснения различий между этими типами помогут вам избавиться от лишних проблем и повысить производительность своей работы.

## 2.1 Примитивные типы в языках программирования

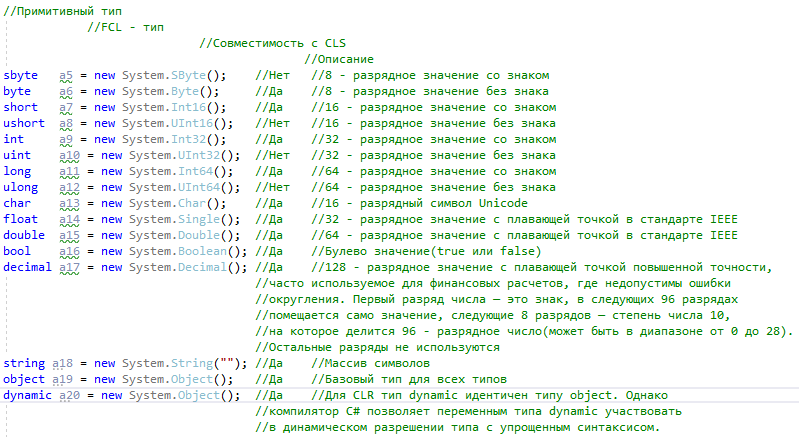
Переменную можно создать следующим образом:

Подобный синтаксис для объявления и инициализации целой переменной кажется громоздким. Многие компиляторы (включая C#) позволяют использовать вместо этого более простые выражения, например: 

Подобный код читается намного лучше, и компилятор в обоих случаях генерирует идентичный IL-код для System.Int32. Типы данных, которые поддерживаются компилятором напрямую, называются примитивными (primitive types); у них существуют прямые аналоги в библиотеке классов .NET Framework Class Library (FCL). Например, типу int языка C# соответствует System.Int32, поэтому весь следующий код компилируется без ошибок и преобразуется в одинаковые IL-команды:



В табл представлены типы FCL и соответствующие им примитивные типы C#. В других языках типам, удовлетворяющим общеязыковой спецификации (Common Language Specification, CLS), соответствуют аналогичные примитивные типы. Однако поддержка языком типов, не удовлетворяющих требованиям CLS, не обязательна.



«С точки зрения стиля программирования предпочтительней использовать ключевое слово, а не полное системное имя типа», поэтому стараюсь задействовать имена FCL-типов и избегать имен примитивных типов.

У многих FCL-типов есть методы, в имена которых включены имена типов. Например, у типа BinaryReader есть методы ReadBoolean, ReadInt32, ReadSingle и т. д., а у типа System.Convert — методы ToBoolean, ToInt32, ToSingle и т. д. Вот вполне приемлемый код, в котором строка, содержащая float, выглядит неестественно; даже возникает впечатление, что код ошибочен:



Многие программисты, пишущие исключительно на С#, часто забывают, что в CLR могут применяться и другие языки программирования. Например, среда FCL практически полностью написана на С#, а разработчики из команды FCL ввели в библиотеку такие методы, как метод GetLongLength класса Array, возвращающий значение Int64, которое имеет тип long в С#, но не в других языках программирования (например, C++/CLI).

По этим причинам я буду использовать в этой книге только имена FCL-типов. Во многих языках программирования следующий код благополучно скомпилируется и выполнится:

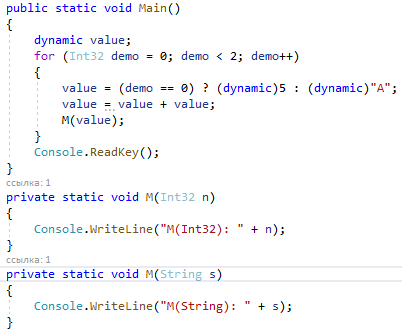


Однако если вспомнить, что говорилось о приведении типов в главе 4, можно решить, что он компилироваться не будет. Все-таки System.Int32 и System.Int64, не являются производными друг от друга. Могу вас обнадежить: код успешно компилируется и делает все, что ему положено. Дело в том, что компилятор C# неплохо разбирается в примитивных типах и применяет свои правила при компиляции кода. Иначе говоря, он распознает наиболее распространенные шаблоны программирования и генерирует такие IL-команды, благодаря которым исходный код работает так, как требуется. В первую очередь, это относится к приведению типов, литералам и операторам, примеры которых мы рассмотрим позже.

### 2.1.1 Примитивный тип данных dynamic

Многие разработчики используют С# для связи с компонентами, не реализованными на С#. Некоторые из этих компонентов могут быть написаны на динамических языках, например Python или Ruby, или быть COM-объектами, которые поддерживают интерфейс IDispatch (возможно, реализованный на С или C++), или объектами модели DOM (Document Object Model), реализованными при помощи разных языков и технологий. Взаимодействие с DOM-объектами особенно полезно для построения Silverlight-приложений.

Для того чтобы облегчить разработку при помощи отражений или коммуникаций с другими компонентами, компилятор С# предлагает помечать типы как динамические (dynamic). Вы также можете записывать результаты вычисления выражений в переменную и пометить ее тип как динамический. Затем динамическое выражение (переменная) может быть использовано для вызовов членов класса, например поля, свойства/индексатора, метода, делегата, или унарных/бинарных операторов. Когда ваш код вызывает член класса при помощи динамического выражения (переменной), компилятор создает специальный IL-код, который описывает желаемую операцию. Этот код называется полезной нагрузкой (payload). Во время выполнения программы он определяет существующую операцию для выполнения на основе действительного типа объекта, на который ссылается динамическое выражение (переменная). Следующий код поясняет, о чем идет речь:



После выполнения метода Main получается следующий результат:

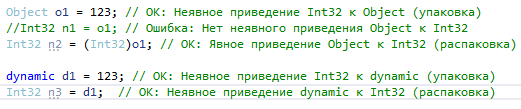
M(Int32): 10

M(String): AA

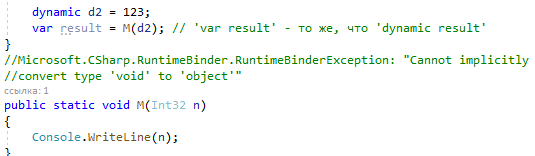
Для того чтобы понять, что здесь происходит, обратимся к оператору +. У этого оператора имеются операнды типа с пометкой dynamic. По этой причине компилятор С# генерирует код полезной нагрузки, который проверяет действительный тип переменной value во время выполнения и определяет, что должен делать оператор +.

Во время первого вызова оператора + значение его аргумента равно 5 (тип Int32), поэтому результат равен 10 (тоже тип Int32). Результат присваивается переменной value. Затем вызывается метод M, которому передается value. Для вызова метода M компилятор создает код полезной нагрузки, который на этапе выполнения будет проверять действительный тип значения переменной, переданной методу M. Когда value содержит тип Int32, вызывается перегрузка метода M, получающая параметр Int32. Во время второго вызова + значение его аргумента равно A (тип String), а результат представляет собой строку AA (результат конкатенации А с собой). Затем снова вызывается метод M, которому передается value. На этот раз код полезной нагрузки определяет, что действительный тип, переданный в M, является строковым, и вызывает перегруженную версию M со строковым параметром. Когда тип поля, параметр метода, возвращаемый тип метода или локальная переменная снабжаются пометкой dynamic, компилятор конвертирует этот тип в тип System.Object и применяет экземпляр System.Runtime.CompilerServices.DynamicAttribute к полю, параметру или возвращаемому типу в метаданных. Если локальная переменная определена как динамическая, то тип переменной также будет типом Object, но атрибут DynamicAttribute неприменим к локальным переменным из-за того, что они используются только внутри метода. Из-за того, что типы dynamic и Object одинаковы, вы не сможете создавать методы с сигнатурами, отличающимися только типами dynamic и Object.

Тип dynamic можно использовать для определения аргументов типов обобщенных классов (ссылочный тип), структур (значимый тип), интерфейсов, делегатов или методов. Когда вы это делаете, компилятор конвертирует тип dynamic в Object и применяет DynamicAttribute к различным частям метаданных, где это необходимо. Обратите внимание, что обобщенный код, который вы используете, уже скомпилирован в соответствии с типом Object, и динамическая отправка не осуществляется, поскольку компилятор не производит код полезной нагрузки в обобщенном коде. Любое выражение может быть явно приведено к dynamic, поскольку все выражения дают в результате тип, производный от Object. В общем случае компилятор не позволит вам написать код с неявным приведением выражения от типа Object к другому типу, вы должны использовать явное приведение типов. Однако компилятор разрешит выполнить приведение типа dynamic к другому типу с использованием синтаксиса неявного приведения.



Пока компилятор позволяет пренебрегать явным приведением динамического типа к другому типу данных, среда CLR на этапе выполнения проверяет правильность приведения с целью обеспечения безопасности типов. Если тип объекта несовместим с приведением, CLR выдает исключение InvalidCastException. Обратите внимание на следующий код:



Здесь компилятор позволяет коду компилироваться, потому что на этапе компиляции он не знает, какой из методов M будет вызван. Следовательно, он также не знает, какой тип будет возвращен методом M. Компилятор предполагает, что переменная result имеет динамический тип. Вы можете убедиться в этом, когда наведете указатель мыши на переменную var в редакторе Visual Studio — во всплывающем IntelliSense-окне вы увидите следующее.dynamic: Represents an object whose operations will be resolved at runtime. Если метод M, вызванный на этапе выполнения, возвращает void, выдается исключение Microsoft.CSharp.RuntimeBinder.RuntimeBinderException.

При преобразовании типа dynamic в другой статический тип результатом будет, очевидно, тоже статический тип. Аналогичным образом при создании типа с передачей конструктору одного и более аргументов dynamic результатом будет объект того типа, который вы создаете:

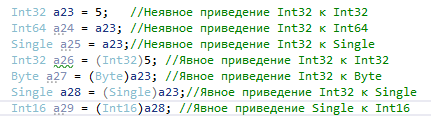


Если выражение dynamic задается как коллекция в инструкции foreach или как ресурс в директиве using, то компилятор генерирует код, который попытается привести выражение к необобщенному интерфейсу System.IEnumerable или интерфейсу System.IDisposable соответственно. Если приведение типов выполняется успешно, то выражение используется, а код выполняется нормально. В противном случае будет выдано исключение Microsoft.CSharp.RuntimeBinder. RuntimeBinderException.

Рассмотрим пример кода на C# с использованием COM-объекта IDispatch для создания книги Microsoft Office Excel и размещения строки в ячейке A1.

## 2.2 Преобразование и приведение типов

Начнем с того, что компилятор выполняет явное и неявное приведение между примитивными типами, например:



C# разрешает неявное приведение типа, если это преобразование «безопасно», то есть не сопряжено с потерей данных; пример — преобразование из Int32 в Int64.

Однако для преобразования с риском потери данных C# требует явного приведения типа. Для числовых типов «небезопасное» преобразование означает «связанное с потерей точности или величины числа». Например, преобразование из Int32 в Byte требует явного приведения к типу, так как при больших величинах Int32 теряется точность; требует приведения и преобразование из Single в Int16, поскольку число Single может оказаться больше, чем допустимо для Int16.

Для реализации приведения разные компиляторы могут порождать разный код. Например, в случае приведения числа 6,8 типа Single к типу Int32 одни компиляторы сгенерируют код, который поместит в Int32 число 6, а другие округлят результат до 7. Между прочим, в C# дробная часть всегда отбрасывается. Точные правила приведения для примитивных типов вы найдете в разделе спецификаций языка C#, посвященном преобразованиям («Conversions»).

## 2.3. Ключевые слова, идентификаторы, литералы

### 2.3.1 Ключевые слова checked и unchecked

Программистам должно быть хорошо известно, что многие арифметические операции над примитивными типами могут привести к переполнению:



Такое «незаметное» переполнение обычно в программировании не приветствуется, и если его не выявить, приложение поведет себя непредсказуемо. Изредка, правда (например, при вычислении хеш-кодов или контрольных сумм), такое переполнение не только приемлемо, но и желательно

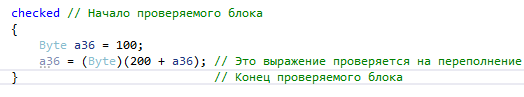
Чтобы включить механизм управления процессом обработки переполнения на этапе компиляции, добавьте в командную строку компилятора параметр /checked+. Он сообщает компилятору, что для выполнения сложения, вычитания, умножения и преобразования должны быть сгенерированы IL-команды с проверкой переполнения. Такой код медленнее, так как CLR тратит время на проверку этих операций, ожидая переполнение. Когда оно возникает, CLR генерирует исключение OverflowException. Код приложения должен предусматривать корректную обработку этого исключения. Однако программистам вряд ли понравится необходимость включения или отключения режима проверки переполнения во всем коде. Им лучше самим решать, как реагировать на переполнение в каждом конкретном случае. И C# предлагает такой механизм гибкого управления проверкой в виде операторов checked и unchecked. Например (предполагается, что компилятор по умолчанию создает код без проверки):



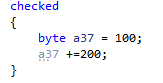
Здесь a34 и 200 преобразуются в 32-разрядные числа и суммируются; результат равен 300. Затем при преобразовании 300 в Byte генерируется исключение OverflowException. Если приведение к типу Byte вывести из оператора checked, исключения не будет:



Наряду с операторами checked и unchecked в C# есть одноименные инструкции, позволяющие включить проверяемые или непроверяемые выражения внутрь блока:



Кстати, внутри такого блока можно задействовать оператор += с Byte, который немного упростит код:



Я видел немало вычислений, генерирующих непредсказуемые результаты. Обычно это случается из-за неправильного ввода данных пользователем или же из-за возвращения неожиданных значений переменных. Итак, я рекомендую программистам соблюдать следующие правила при использовании операторов checked и unchecked.

Используйте типы со знаком (Int32 и Int64) вместо числовых типов без знака (UInt32 и UInt64) везде, где это возможно. Это позволит компилятору выявлять ошибку переполнения. Кроме того, некоторые компоненты библиотеки классов (например, свойства Length классов Array и String) жестко запрограммированы на возвращение значений со знаком, и передача этих значений в коде потребует меньшего количества преобразований типа (а следовательно, упростит структуру кода и его сопровождение). Кроме того, числовые типы без знака несовместимы с CLS.

Включайте в блок checked ту часть кода, в которой возможно переполнение из-за неверных входных данных, например при обработке запросов, содержащих данные, предоставленные конечным пользователем или клиентской машиной. Возможно, также стоит перехватывать исключение OverflowException, чтобы ваше приложение могло корректно продолжить работу после таких сбоев.

Включайте в блок unchecked те фрагменты кода, в которых переполнение не создает проблем (например, при вычислении контрольной суммы).

В коде, где нет операторов и блоков checked и unchecked, предполагается, что при переполнении должно происходить исключение. Например, при вычислении простых чисел входные данные известны, а переполнение является признаком ошибки.

#### 2.3.1.1 Про тип Decimal и BigInteger

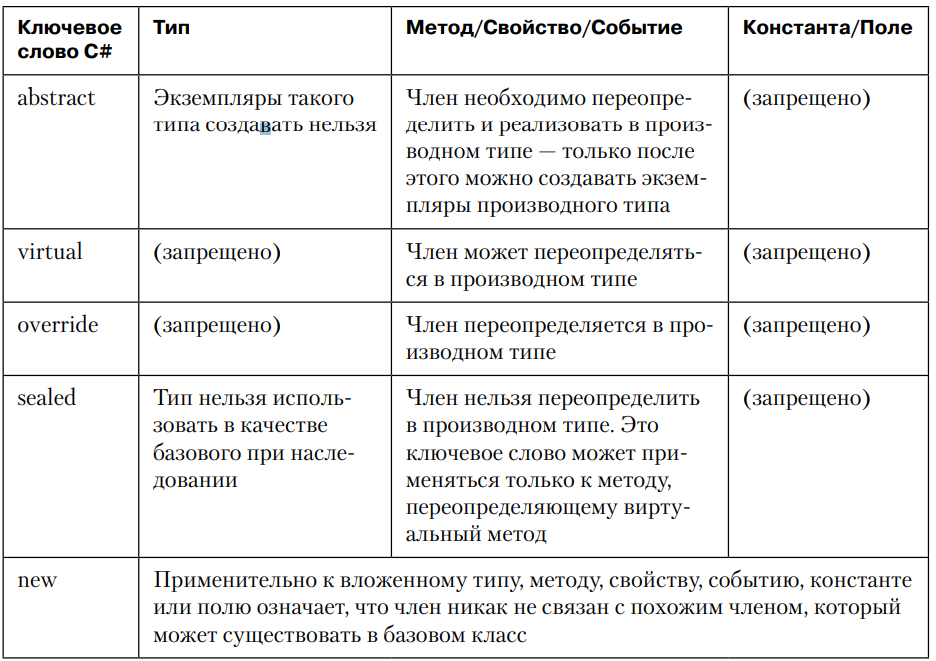
Тип System.Decimal стоит особняком. В отличие от многих языков программирования (включая C# и Visual Basic), в CLR тип Decimal не относится к примитивным типам. В CLR нет IL-команд для работы со значениями типа Decimal. В документации по .NET Framework сказано, что тип Decimal имеет открытые статические методы-члены Add, Subtract, Multiply, Divide и прочие, а также перегруженные операторы +, -, \*, / и т. д. При компиляции кода с типом Decimal компилятор генерирует вызовы членов Decimal, которые и выполняют реальную работу. Поэтому значения типа Decimal обрабатываются медленнее примитивных CLR-типов. Кроме того, раз нет IL-команд для манипуляции числами типа Decimal, то не будут иметь эффекта ни операторы checked и unchecked, ни соответствующие параметры командной строки компилятора, а неосторожность в операциях над типом Decimal может привести к исключению OverflowException. Аналогично, тип System.Numerics.BigInteger используется в массивах UInt32 для представления большого целочисленного значения, не имеющего верхней или нижней границы. Следовательно, операции с типом BigInteger никогда не вызовут исключения OverflowException. Однако они могут привести к выдаче исключения OutOfMemoryException, если значение переменной окажется слишком большим

### 2.3.2 Ключевое слово var

Объявление локальной переменной как var является синтаксическим указанием компилятору подставлять специальные данные из соответствующего выражения. Ключевое слово var может использоваться только для объявления локальных переменных внутри метода, тогда как ключевое слово dynamic может указываться с локальными переменными, полями и аргументами. Вы не можете привести выражение к типу var, но вы можете привести его к типу dynamic. Вы должны явно инициализировать переменную, объявленную как var, тогда как переменную, объявленную как dynamic, инициализировать нельзя. Больше подробно о типе var рассказывается в главе 9.

### 2.3.3 Ключевое слово static(нет ответа)

### 2.3.4 Ключевые слова C#, относящиеся к управлению версиями



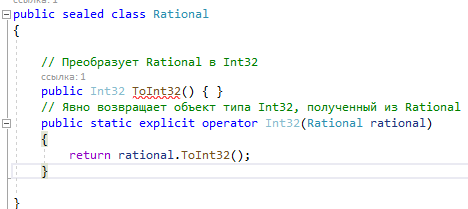
### 2.3.6 Ключевое слово implicit

Ключевое слово implicit указывает компилятору C#, что наличие в исходном тексте явного приведения типов не обязательно для генерации кода, вызывающего метод оператора преобразования.



### 2.3.7 Ключевое слово explicit

Ключевое слово explicit позволяет компилятору вызывать метод только тогда, когда в исходном тексте происходит явное приведение типов.



### 2.3.8 Ключевое слово default (нет ответа)

### 2.3.9 Ключевое слово ref или out (нет ответа)

### 2.3.10 Литералы

В C# литералами называются постоянные значения, представленные в удобной для восприятия форме. Например, число 100 является литералом. Сами литералы и их назначение настолько понятны, что они применялись во всех предыдущих примерах программ без всяких пояснений. Но теперь настало время дать им формальное объяснение.

В C# литералы могут быть любого простого типа. Представление каждого литерала зависит от конкретного типа. Как пояснялось ранее, символьные литералы заключаются в одинарные кавычки. Например, 'а' и '%' являются символьными литералами.

Помимо приведения, компилятор «знает» и о другой особенности примитивных типов: к ним применима литеральная форма записи. Литералы сами по себе считаются экземплярами типа, поэтому можно вызывать экземплярные методы, например, следующим образом: 

Кроме того, благодаря тому, что выражения, состоящие из литералов, вычисляются на этапе компиляции, возрастает скорость выполнения приложения.



И наконец, компилятор «знает», как и в каком порядке интерпретировать встретившиеся в коде операторы (в том числе +, -, \*, /, %, &, ^, |, ==, !=, >, <, >=, <=, <<, >>, ~, !, ++, -- и т. п.):



## 2.4 Ссылочные и значимые типы

CLR поддерживает две разновидности типов: ссылочные (reference types) и значимые (value types). Большинство типов в FCL — ссылочные, но программисты чаще всего используют значимые. Память для ссылочных типов всегда выделяется из управляемой кучи, а оператор C# new возвращает адрес в памяти, где размещается сам объект. При работе со ссылочными типами необходимо учитывать следующие обстоятельства, относящиеся к производительности приложения:

память для ссылочных типов всегда выделяется из управляемой кучи;

каждый объект, размещаемый в куче, содержит дополнительные члены, подлежащие инициализации;

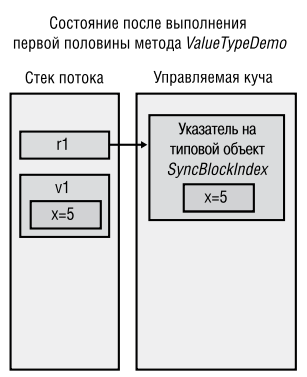
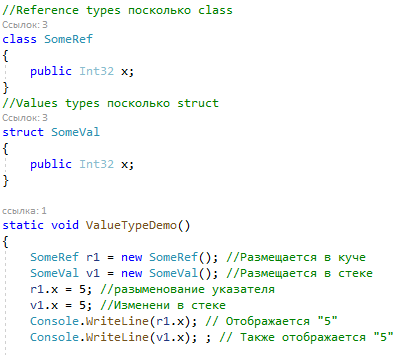
незанятые полезной информацией байты объекта обнуляются (это касается полей);

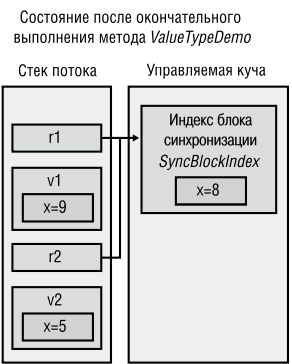
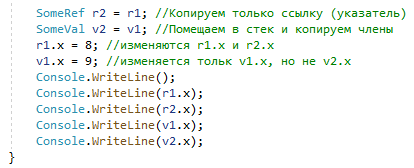
размещение объекта в управляемой куче со временем инициирует сборку мусора.

Если бы все типы были ссылочными, эффективность приложения резко упала бы. Представьте, насколько замедлилось бы выполнение приложения, если бы при каждом обращении к значению типа Int32 выделялась память! Поэтому, чтобы ускорить обработку простых, часто используемых типов CLR предлагает «облегченные» типы — значимые. Экземпляры этих типов обычно размещаются в стеке потока (хотя они могут быть встроены и в объект ссылочного типа). В представляющей экземпляр переменной нет указателя на экземпляр; поля экземпляра размещаются в самой переменной. Поскольку переменная содержит поля экземпляра, то для работы с экземпляром не нужно выполнять разыменование (dereference) экземпляра. Благодаря тому, что экземпляры значимых типов не обрабатываются уборщиком мусора, уменьшается интенсивность работы с управляемой кучей и сокращается количество сеансов уборки мусора, необходимых приложению на протяжении его существования.

В документации на .NET Framework можно сразу увидеть, какие типы относятся к ссылочным, а какие — к значимым. Если тип называют классом (class), речь идет о ссылочном типе. Например, классы System.Object, System.Exception, System.IO.FileStream и System.Random — это ссылочные типы. В свою очередь, значимые типы в документации называются структурами (structure) и перечислениями (enumeration). Например, структуры System.Int32, System.Boolean, System.Decimal, System.TimeSpan и перечисления System.DayOfWeek, System. IO.FileAttributes и System.Drawing.FontStyle являются значимыми типами. Все структуры являются прямыми потомками абстрактного типа System.ValueType, который, в свою очередь, является производным от типа System. Object. По умолчанию все значимые типы должны быть производными от System. ValueType. Все перечисления являются производными от типа System.Enum, производного от System.ValueType. При определении собственного значимого типа нельзя выбрать произвольный базовый тип, однако значимый тип может реализовать один или несколько выбранных вами интерфейсов. Кроме того, в CLR значимый тип является изолированным, то есть он не может служить базовым типом для какого-либо другого ссылочного или значимого типа. Поэтому, например, нельзя в описании нового типа указывать в качестве базовых типы Boolean, Char, Int32, Uint64, Single, Double, Decimal и т. д.

В следующем коде продемонстрировано различие между ссылочными и значимыми типами:





В этом примере тип SomeVal объявлен с ключевым словом struct, а не более распространенным ключевым словом class. В C# типы, объявленные как struct, являются значимыми, а объявленные как class, — ссылочными. Между поведением ссылочных и значимых типов существуют существенные различия. Поэтому так важно представлять, к какому семейству относится тот или иной тип — к ссылочному или значимому: ведь это может существенно повлиять на то, как вы выражаете свои намерения в коде. В предыдущем примере есть следующая строка:



Может показаться, что экземпляр SomeVal будет помещен в управляемую кучу. Однако поскольку компилятор C# «знает», что SomeVal является значимым типом, в сгенерированном им коде экземпляр SomeVal будет помещен в стек потока. C# также обеспечивает обнуление всех полей экземпляра значимого типа. Ту же строку можно записать иначе:

SomeVal v1; // Размещается в стеке

Здесь тоже создается IL-код, который помещает экземпляр SomeVal в стек потока и обнуляет все его поля. Единственное отличие в том, что экземпляр, созданный оператором new, C# «считает» инициализированным. Поясню эту мысль на следующем примере:

Две следующие строки компилируются, так как C# считает, что поля в v1 инициализируются нулем

SomeVal v1 = new SomeVal();

Int32 a = v1.x;

Следующие строки вызовут ошибку компиляции, поскольку C# не считает, что поля в v1 инициализируются нулем

SomeVal v1;

Int32 a = v1.x;

// error CS0170: Use of possibly unassigned field 'x'

// (ошибка CS0170: Используется поле 'x', которому не присвоено значение)

Проектируя свой тип, проверьте, не использовать ли вместо ссылочного типа значимый. Иногда это позволяет повысить эффективность кода. Сказанное особенно справедливо для типа, удовлетворяющего всем перечисленным далее условиям.

Тип ведет себя подобно примитивному типу. В частности, это означает, что тип достаточно простой и у него нет членов, способных изменить экземплярные поля типа, в этом случае говорят, что тип неизменяемый (immutable). На самом деле, многие значимые типы рекомендуется помечать спецификатором readonly (см. главу 7).

Тип не обязан иметь любой другой тип в качестве базового.

Тип не имеет производных от него типов. Также необходимо учитывать размер экземпляров типа, потому что по умолчанию аргументы передаются по значению; при этом поля экземпляров значимого типа копируются, что отрицательно сказывается на производительности. Повторюсь: для метода, возвращающего значимый тип, поля экземпляра копируются в память, выделенную вызывающим кодом в месте возврата из метода, что снижает эффективность работы программы. Поэтому в дополнение к перечисленным условиям следует объявлять тип как значимый, если верно хотя бы одно из следующих условий:

Размер экземпляров типа мал (примерно 16 байт или меньше).

Размер экземпляров типа велик (более 16 байт), но экземпляры не передаются в качестве параметров метода или не являются возвращаемыми из метода значениями. Основное достоинство значимых типов в том, что они не размещаются в управляемой куче. Конечно, в сравнении со ссылочными типами у значимых типов есть недостатки. Важнейшие отличия между значимыми и ссылочными типы:

Объекты значимого типа существуют в двух формах (см. следующий раздел): неупакованной (unboxed) и упакованной (boxed). Ссылочные типы бывают только в упакованной форме.

Значимые типы являются производными от System.ValueType. Этот тип имеет те же методы, что и System.Object. Однако System.ValueType переопределяет метод Equals, который возвращает true, если значения полей в обоих объектах совпадают. Кроме того, в System.ValueType переопределен метод GetHashCode, который создает хеш-код по алгоритму, учитывающему значения полей экземпляра объекта. Из-за проблем с производительностью в реализации по умолчанию, определяя собственные значимые типы значений, надо переопределить и написать свою реализацию методов Equals и GetHashCode. О методах Equals и GetHashCode рассказано в конце этой главы.

Поскольку в объявлении нового значимого или ссылочного типа нельзя указывать значимый тип в качестве базового класса, создавать в значимом типе новые виртуальные методы нельзя. Методы не могут быть абстрактными и неявно являются запечатанными (то есть их нельзя переопределить).

Переменные ссылочного типа содержат адреса объектов в куче. Когда переменная ссылочного типа создается, ей по умолчанию присваивается null, то есть в этот момент она не указывает на действительный объект. Попытка задействовать переменную с таким значением приведет к генерации исключения NullReferenceException. В то же время в переменной значимого типа всегда содержится некое значение соответствующего типа, а при инициализации всем членам этого типа присваивается 0. Поскольку переменная значимого типа не является указателем, при обращении к значимому типу исключение NullReferenceException возникнуть не может. CLR поддерживает понятие значимого типа особого вида, допускающего присваивание null (nullable types). Этот тип обсуждается в главе 19.

Когда переменной значимого типа присваивается другая переменная значимого типа, выполняется копирование всех ее полей. Когда переменной ссылочно-го типа присваивается переменная ссылочного типа, копируется только ее адрес.

Вследствие сказанного в предыдущем пункте несколько переменных ссылочного типа могут ссылаться на один объект в куче, благодаря чему, работая с одной переменной, можно изменить объект, на который ссылается другая переменная. В то же время каждая переменная значимого типа имеет собственную копию данных «объекта», поэтому операции с одной переменной значимого типа не влияют на другую переменную.

Так как неупакованные значимые типы не размещаются в куче, отведенная для них память освобождается сразу при возвращении управления методом, в котором описан экземпляр этого типа (в отличие от ожидания уборки мусора).

### 2.4.1 Члены типа

В типе можно определить следующие члены.

Константа — идентификатор, определяющий некую постоянную величину. Эти идентификаторы обычно используют, чтобы упростить чтение кода, а также для удобства сопровождения и поддержки. Константы всегда связаны с типом, а не с экземпляром типа, а на логическом уровне константы всегда являются статическими членами. Подробнее о константах см. главу 7.

Поле представляет собой значение данных, доступное только для чтения или для чтения/записи. Поле может быть статическим — тогда оно является частью состояния типа. Поле может быть экземплярным (нестатическим) — тогда оно является частью состояния конкретного объекта. Я настоятельно рекомендую ограничивать доступ к полям, чтобы внешний код не мог нарушить состояние типа или объекта. Подробнее о полях см. главу 7.

Конструктор экземпляров — метод, служащий для инициализации полей экземпляра при его создании. Подробнее о конструкторах экземпляров см. главу 8.

Конструктор типа — метод, используемый для инициализации статических полей типа. Подробнее о конструкторах типа см. главу 8.

Метод представляет собой функцию, выполняющую операции, которые изменяют или запрашивают состояние типа (статический метод) или объекта (экземплярный метод). Методы обычно осуществляют чтение и запись полей типов или объектов. Подробнее о методах см. главу 8.

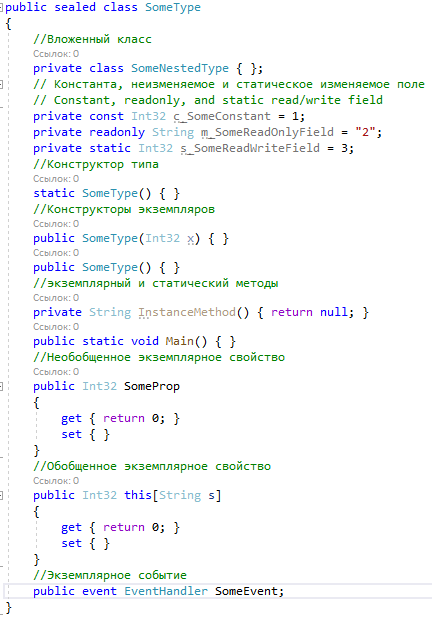
Перегруженный оператор определяет, что нужно проделать с объектом при применении к нему конкретного оператора. Перегрузка операторов не входит в общеязыковую спецификацию CLS, поскольку не все языки программирования ее поддерживают. Подробнее о перегруженных операторах см. главу 8.

Оператор преобразования — метод, задающий порядок явного или неявного преобразования объекта из одного типа в другой. Операторы преобразования не входят в спецификацию CLS по той же причине, что и перегруженные операторы. Подробнее об операторах преобразования см. главу 8.

Свойство представляет собой механизм, позволяющий применить простой синтаксис (напоминающий обращение к полям) для установки или получения части логического состояния типа или объекта с контролем логической целостности этого состояния. Свойства бывают необобщенными (распространенный случай) и обобщенными (встречаются редко, в основном в классах коллекций). Подробнее о свойствах см. главу 10.

Событие — механизм статических событий позволяет типу отправлять уведомления статическим или экземплярным методам. Механизм экземплярных (нестатических) событий позволяет объекту посылать уведомление статическому или экземплярному методу. События обычно инициируются в ответ на изменение состояния типа или объекта, порождающего событие. Событие состоит из двух методов, позволяющих статическим или экземплярным методам регистрировать и отменять регистрацию (подписку) на событие. Помимо этих двух методов, в событиях обычно используется поле-делегат для управления набором зарегистрированных методов. Подробнее о событиях см. главу 11.

Тип позволяет определять другие вложенные в него типы. Обычно этот подход применяется для разбиения большого, сложного типа на небольшие блоки с целью упростить его реализацию.

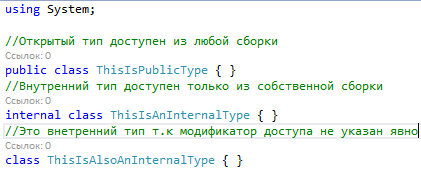


Независимо от используемого языка программирования, компилятор должен обработать исходный код и создать метаданные и IL-код для всех членов типа. Формат метаданных един и не зависит от выбранного языка программирования — именно поэтому CLR называют общеязыковой исполняющей средой. Метаданные — это стандартная информация, которую предоставляют и используют все языки, позволяя коду на одном языке программирования без проблем обращаться к коду на совершенно другом языке. Стандартный формат метаданных также используется средой CLR для определения порядка поведения констант, полей, конструкторов, методов, свойств и событий во время выполнения. Короче говоря, метаданные — это ключ ко всей платформе разработки Microsoft .NET Framework; они обеспечивают интеграцию языков, типов и объектов. В следующем примере на C# показано определение типа со всеми возможными членами. Этот код успешно компилируется (не без предупреждений), но пользы от него немного — он всего лишь демонстрирует, как компилятор транслирует такой тип и его члены в метаданные.

После компиляции типа можно просмотреть метаданные с помощью утилиты ILDasm.exe (рис. 6.1). Заметьте, что компилятор генерирует метаданные для всех членов типа. На самом деле, для некоторых членов, например для события (17), компилятор создает дополнительные члены (поле и два метода) и метаданные.

### 2.4.2 Видимость типа

При определении типа с видимостью в рамках файла, а не другого типа его можно сделать открытым (public) или внутренним (internal). Открытый тип доступен любому коду любой сборки. Внутренний тип доступен только в той сборке, где он определен. По умолчанию компилятор C# делает тип внутренним (с более ограниченной видимостью). Вот несколько примеров.

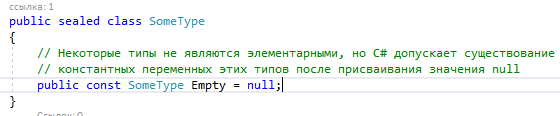


### 2.4.3 Дружественные сборки (не понял)

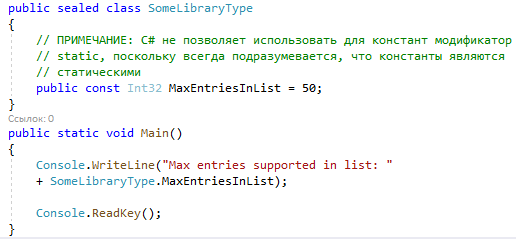
## 2.5 Выражения и операции(нет ответа)

## 2.6 Константы

Константа (constant) — это идентификатор, значение которого никогда не меняется. Значение, связанное с именем константы, должно определяться во время компиляции. Затем компилятор сохраняет значение константы в метаданных модуля. Это значит, что константы можно определять только для таких типов, которые компилятор считает примитивными. В C# следующие типы считаются примитивными и могут использоваться для определения констант: Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal и String. Тем не менее C# позволяет определить константную переменную, не относящуюся к элементарному типу, если присвоить ей значение null:



Так как значение констант никогда не меняется, константы всегда считаются частью типа. Иначе говоря, константы считаются статическими, а не экземплярными членами. Определение константы приводит в конечном итоге к созданию метаданных. Встретив в исходном тексте имя константы, компилятор просматривает метаданные модуля, в котором она определена, извлекает значение константы и внедряет его в генерируемый им IL-код. Поскольку значение константы внедряется прямо в код, в период выполнения память для констант не выделяется. Кроме того, нельзя получать адрес константы и передавать ее по ссылке. Эти ограничения также означают, что изменять значения константы в разных версиях модуля нельзя, поэтому константу надо использовать, только когда точно известно, что ее значение никогда не изменится (хороший пример — определение константы MaxInt16 со значением 32767). Поясню на примере, что я имею в виду. Возьмем код и скомпилируем его в DLL-сборку:

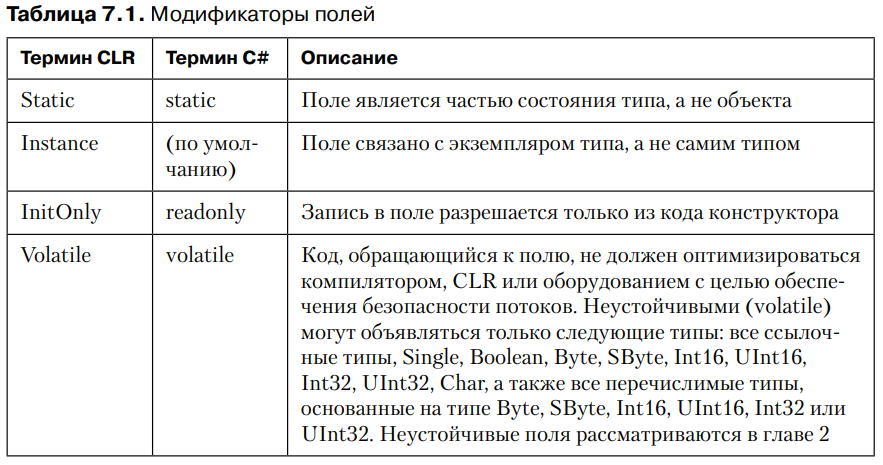


Нетрудно заметить, что код приложения содержит ссылку на константу MaxEntriesInList. При компоновке этого кода компилятор, обнаружив, что MaxEntriesInList — это литерал константы со значением 50, внедрит значение 50 типа Int32 прямо в IL-код приложения. Фактически после построения кода приложения DLL-сборка даже не будет загружаться в период выполнения, поэтому ее можно просто удалить с диска.

Теперь проблема управления версиями при использовании констант должна стать очевидной. Если разработчик изменит значение константы MaxEntriesInList на 1000 и перестроит только DLL-сборку, это не повлияет на код самого приложения. Для того чтобы в приложении использовалось новое значение константы, его тоже необходимо перекомпилировать. Нельзя применять константы во время выполнения (а не во время компиляции), если модуль должен задействовать значение, определенное в другом модуле. В этом случае вместо констант следует использовать предназначенные только для чтения поля, о которых речь идет в следующем разделе.

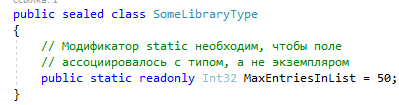
## 2.7 Поля

Поле (field) — это член данных, который хранит экземпляр значимого типа или ссылку на ссылочный тип. В табл. 7.1 приведены модификаторы, применяемые по отношению к полям.

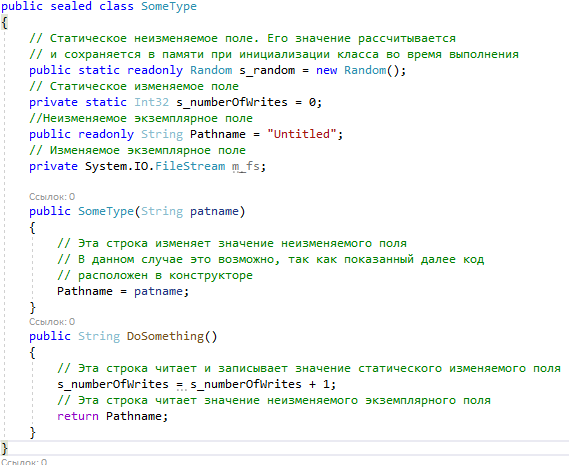


Как видно из таблицы, общеязыковая среда (CLR) поддерживает поля как типов (статические), так и экземпляров (нестатические). Динамическая память для хранения поля типа выделяется в пределах объекта типа, который создается при загрузке типа в домен приложений (см. главу 22), что обычно происходит при JIT-компиляции любого метода, ссылающегося на этот тип. Динамическая память для хранения экземплярных полей выделяется при создании экземпляра данного типа.

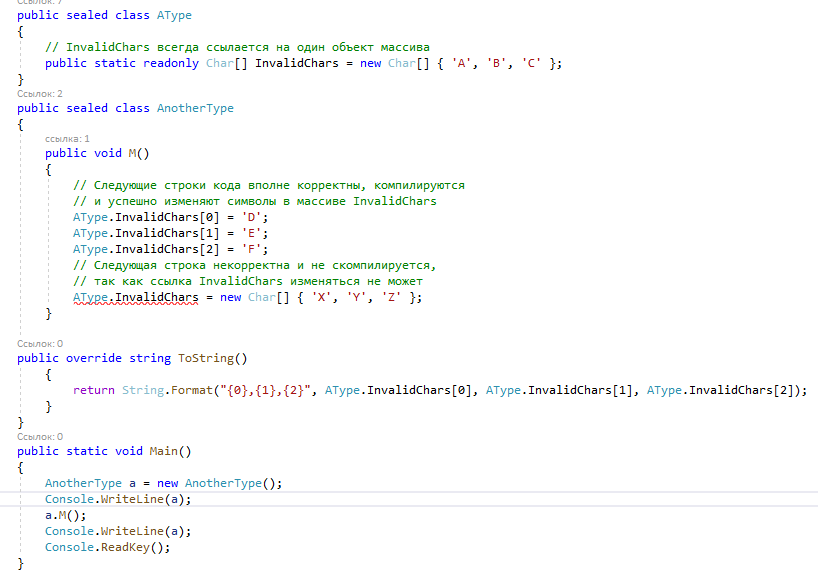
Поскольку поля хранятся в динамической памяти, их значения можно получить лишь в период выполнения. Поля также решают проблему управления версиями, возникающую при использовании констант. Кроме того, полю можно назначить любой тип данных, поэтому при определении полей можно не ограничиваться встроенными элементарными типами компилятора (что приходится делать при определении констант). CLR поддерживает поля, предназначенные для чтения и записи (изменяемые), а также поля, предназначенные только для чтения (неизменяемые). Большинство полей изменяемые. Это значит, что во время исполнения кода значение таких полей может многократно меняться. Данные же в неизменяемые поля можно записывать только при исполнении конструктора (который вызывается лишь раз — при создании объекта). Компилятор и механизм верификации гарантируют, что ни один метод, кроме конструктора, не сможет записать данные в поле, предназначенное только для чтения. Замечу, что для изменения такого поля можно задействовать отражение. Попробуем решить проблему управления версиями в примере из раздела «Константы», используя статические неизменяемые поля. Вот новая версия кода DLL-сборки:



Это единственное изменение, которое придется внести в исходный текст, при этом код приложения можно вовсе не менять, но чтобы увидеть его новые свойства, его придется перекомпилировать. Теперь при исполнении метода Main этого приложения CLR загружает DLL-сборку (так как она требуется во время выполнения) и извлекает значение поля MaxEntriesInList из динамической памяти, выделенной для его хранения. Естественно, это значение равно 50. Допустим, разработчик сборки изменил значение поля с 50 на 1000 и скомпоновал сборку заново. При повторном исполнении код приложения автоматически задействует новое значение — 1000. В этом случае не обязательно компоновать код приложения заново, оно просто работает в том виде, в котором было (хотя и чуть медленнее). Однако этот сценарий предполагает, что у новой сборки нет строгого имени, а политика управления версиями приложения заставляет CLR загружать именно эту новую версию сборки. В следующем примере показано, как определять изменяемые статические поля, а также изменяемые и неизменяемые экземплярные поля:



Многие поля в нашем примере инициализируются на месте (inline). C# позволяет использовать этот удобный синтаксис для инициализации констант, а также изменяемых и неизменяемых полей. Как продемонстрировано в главе 8, C# рассматривает инициализацию поля на месте как синтаксис сокращенной записи, позволяющий инициализировать поле во время исполнения конструктора. Вместе с тем, в C# возможны проблемы производительности, которые нужно учитывать при использовании синтаксиса инициализации поля на месте, а не присвоения в конструкторе. Они также обсуждаются в главе 8.

Внимание Неизменность поля ссылочного типа означает неизменность ссылки, которую этот тип содержит, а вовсе не объекта, на которую указывает ссылка, например:

## 2.7 Переменные (нет ответа)

# 3. Модификаторы доступа

Модификаторы определяют, на какие члены можно ссылаться из кода. В CLR имеется собственный набор возможных модификаторов доступа, но в каждом языке программирования существуют свои синтаксис и термины. Например, термин Assembly в CLR указывает, что член доступен изнутри сборки, тогда как в C# для этого используется ключевое слово internal.

## 3.1 Доступ к членам типов

При определении члена типа (в том числе вложенного) можно указать модификатор доступа к члену.private (закрытый) - Доступен только методам в определяющем типе и вложенных в него типах;

protected (родовой) - Доступен только методам в определяющем типе и вложенных в него типах или в одном из его производных типов независимо от сборки;

internal (сборочный) - Доступен только методам в определяющей сборке;

protected internal (сборочный или родовой) - Доступен только методам вложенного типа, производного типа (независимо от сборки) и любым методам определяющей сборки;

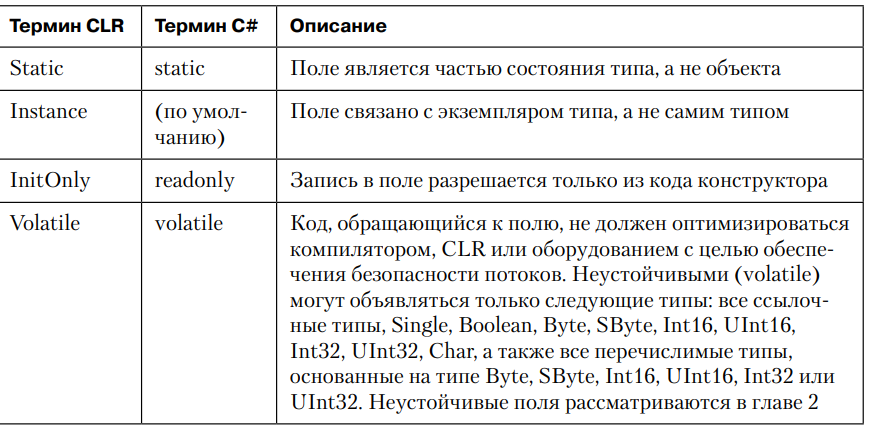
private protected - такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке;

public - Доступен всем методам во всех сборках.

Если модификатор доступа явно не указан, компилятор C# обычно (но не всегда) выберет по умолчанию закрытый — наиболее строгий из всех. CLR требует, чтобы все члены интерфейсного типа были открытыми. Компилятор C# знает об этом и запрещает программисту явно указывать модификаторы доступа к членам интерфейса, просто делая все члены открытыми.

Более того, как видно из табл. 6.1, в CLR есть модификатор доступа родовой и сборочный. Но разработчики C# сочли этот атрибут лишним и не включили в язык C#. Если в производном типе переопределяется член базового типа, компилятор C# требует, чтобы у членов базового и производного типов были одинаковые модификаторы доступа. То есть если член базового класса является защищенным, то и член производного класса должен быть защищенным. Однако это ограничение языка C#, а не CLR. При наследовании от базового класса CLR позволяет снижать, но не повышать ограничения доступа к члену. Например, защищенный метод базового класса можно переопределить в производном классе в открытый, но не в закрытый. Дело в том, что пользователь производного класса всегда может получить доступ к методу базового класса путем приведения к базовому типу. Если бы среда CLR разрешала устанавливать более жесткие ограничения на доступ к методу в производном типе, то эти ограничения бы элементарно обходились.

## 3.2 Модификаторы доступа полей



# 4. Классы

## 4.1 Статические классы

Существуют классы, не предназначенные для создания экземпляров, например Console, Math, Environment и ThreadPool. У этих классов есть только статические методы. В сущности, такие классы существуют лишь для группировки логически связанных членов. Например, класс Math объединяет методы, выполняющие математические операции. В C# такие классы определяются с ключевым словом static. Его разрешается применять только к классам, но не к структурам (значимым типам), поскольку CLR всегда разрешает создавать экземпляры значимых типов, и нет способа обойти это ограничение. Компилятор налагает на статический класс ряд ограничений.

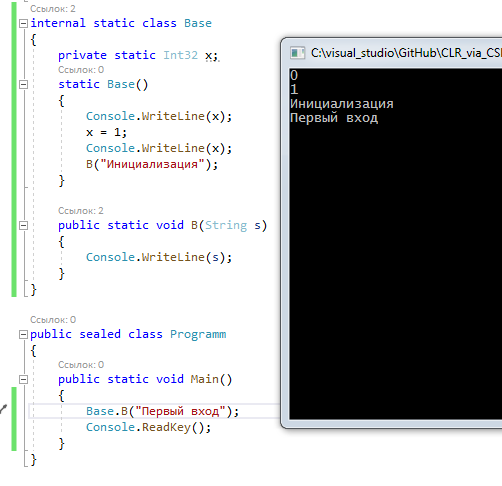
Класс должен быть прямым потомком System.Object — наследование любому другому базовому классу лишено смысла, поскольку наследование применимо только к объектам, а создать экземпляр статического класса невозможно.

Класс не должен реализовывать никаких интерфейсов, поскольку методы интерфейса можно вызывать только через экземпляры класса.

В классе можно определять только статические члены (поля, методы, свойства и события). Любые экземплярные члены вызовут ошибку компиляции.

Класс нельзя использовать в качестве поля, параметра метода или локальной переменной, поскольку это подразумевает существование переменной, ссылающейся на экземпляр, что запрещено. Обнаружив подобное обращение со статическим классом, компилятор вернет сообщение об ошибке.

Приведем пример статического класса, в котором определены статические члены; сам по себе класс не представляет практического интереса.



## 4.2 Частичные классы

Ключевое слово partial говорит компилятору C#, что исходный код класса, структуры или интерфейса может располагаться в нескольких файлах. Компилятор объединяет все частичные файлы класса во время компиляции; CLR всегда работает с полными определениями типов. Есть три основные причины, по которым исходный код разбивается на несколько файлов.

Управление версиями. Представьте, что определение типа содержит большой объем исходного кода. Если этот тип будут одновременно редактировать два программиста, по завершении работы им придется каким-то образом объединять свои результаты, что весьма неудобно. Ключевое слово partial позволяет разбить исходный код типа на несколько файлов, чтобы один и тот же тип могли одновременно редактировать несколько программистов.

Разделение файла или структуры на логические модули внутри файла. Иногда требуется создать один тип для решения разных задач. Для упрощения реализации я иногда объявляю одинаковые типы повторно внутри одного файла. Затем в каждой части такого типа я реализую по одному функциональному аспекту типа со всеми его полями, методами, свойствами, событиями и т. д. Это позволяет мне упростить наблюдение за членами, обеспечивающими единую функциональность и объединенными в группу. Я также могу легко закомментировать частичный тип с целью удаления всей функциональности из класса или замены ее другой (путем использования новой части частичного типа).

Разделители кода. При создании в Microsoft Visual Studio нового проекта Windows Forms или Web Forms некоторые файлы с исходным кодом создаются автоматически. Они называются шаблонными. При использовании конструкторов форм Visual Studio в процессе создания и редактирования элементов управления формы автоматически генерирует весь необходимый код и помещает его в отдельные файлы. Это значительно повышает продуктивность работы. Раньше автоматически генерируемый код попадал в тот же файл, где программист писал свой исходный код. Однако при случайном изменении сгенерированного кода конструктор форм переставал корректно работать. Начиная с Visual Studio 2005, при создании нового проекта Visual Studio создает два исходных файла: один предназначен для программиста, а в другой помещается код, создаваемый редактором форм. В результате вероятность случайного изменения генерируемого кода существенно снижается.

Ключевое слово partial применяется к типам во всех файлах с определением типа. При компиляции компилятор объединяет эти файлы, и готовый тип помещается в результирующий файл сборки с расширением exe или dll (или в файл модуля с расширением netmodule). Как уже отмечалось, частичные типы реализуются только компилятором C#; поэтому все файлы с исходным кодом таких типов необходимо писать на одном языке и компилировать их вместе как единый блок компиляции.

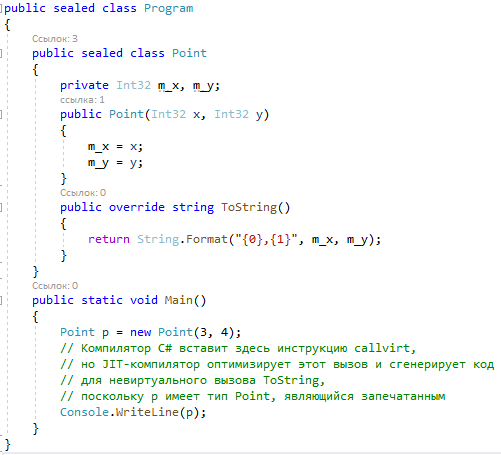
## 4.3 Запечатанные классы

Разумное использование видимости типов и модификаторов доступа к членам.

В .NET Framework приложения состоят из типов, определенных в многочисленных сборках, созданных различными компаниями. Это означает практически полное отсутствие контроля над используемыми компонентами и типами. Разработчику обычно недоступен исходный код компонентов (он может даже не знать, на каком языке они написаны), к тому же версии компонентов обновляются в разное время. Более того, из-за полиморфизма и наличия защищенных членов разработчик базового класса должен доверять коду разработчика производного класса. В свою очередь, разработчик производного класса должен доверять коду, наследуемому от базового класса. В этом разделе я расскажу о том, как проектировать типы с учетом этих факторов. А если говорить конкретно, речь пойдет о том, как правильно задавать видимость типов и модификаторы доступа к членам. В первую очередь при определении нового типа компиляторам следовало бы по умолчанию делать его запечатанным. Вместо этого большинство компиляторов (в том числе C#) поступают как раз наоборот, считая, что программист при необходимости сам может запечатать класс с помощью ключевого слова sealed. Есть три веские причины в пользу использования запечатанных классов.

Управление версиями. Если класс изначально запечатан, его впоследствии можно сделать незапечатанным, не нарушая совместимости. Однако обратное невозможно, поскольку это нарушило бы работу всех производных классов. Кроме того, если в незапечатанном классе определены незапечатанные виртуальные методы, необходимо сохранять порядок вызова виртуальных методов в новых версиях, иначе в будущем возникнут проблемы с производными типами.

Производительность. Как уже отмечалось, невиртуальные методы вызываются быстрее виртуальных, поскольку для последних CLR во время выполнения проверяет тип объекта, чтобы выяснить, где находится метод. Однако, встретив вызов виртуального метода в запечатанном типе, JIT-компилятор может сгенерировать более эффективный код, задействовав невиртуальный вызов. Это возможно потому, что у запечатанного класса не может быть производных классов. Например, в следующем коде JIT-компилятор может вызвать виртуальный метод ToString невиртуально:



Безопасность и предсказуемость. Состояние класса должно быть надежно защищено. Если класс не запечатан, производный класс может изменить его состояние, воспользовавшись незащищенными полями или методами базового класса, изменяющими его доступные незакрытые поля. Кроме того, в производном классе можно переопределить виртуальные методы и не вызывать реализацию соответствующих методов базового класса. Назначая метод, свойство и событие виртуальным, базовый класс уступает некоторую степень контроля над его поведением и состоянием производному классу, что при неумелом обращении может вызвать непредсказуемое поведение и проблемы с безопасностью.

Беда в том, что запечатанные классы могут создать изрядные неудобства для пользователей типа. Разработчику приложения может понадобиться производный тип, в котором будут добавлены дополнительные поля или другая информация о состоянии. Они даже могут попытаться добавить в производном типе дополнительные методы для работы с этими полями. Хотя CLR не предоставляет механизма расширения уже построенных типов вспомогательными методами или полями, вспомогательные методы можно имитировать при помощи методов расширения C# (см. главу 8), а для расширения состояния объекта может использоваться класс ConditionalWeakTable (см. главу 21). Вот несколько правил, которым я следую при проектировании классов:

Если класс не предназначен для наследования, я всегда явно объявляю его запечатанным. Как уже отмечалось, C# и многие современные компиляторы поступают иначе. Если нет необходимости в предоставлении другим сборкам доступа к классу, он объявляется внутренним. К счастью, именно так ведет себя по умолчанию компилятор C#. Если я хочу определить класс, предназначенный для создания производных классов, одновременно запретив его специализацию, я должен переопределить и запечатать все виртуальные методы, которые наследует мой класс.

Все поля данных класса всегда объявляются закрытыми, и в этом я никогда не уступлю. К счастью, по умолчанию C# поступает именно так. Вообще говоря, я бы предпочел, чтобы в C# остались только закрытые поля, а объявлять их со спецификаторами protected, internal, public и т. д. было бы запрещено. Доступ к состоянию объекта — верный путь к непредсказуемому поведению и проблемам с безопасностью. При объявлении полей внутренними (internal) также могут возникнуть проблемы, поскольку даже внутри одной сборки очень трудно отследить все обращения к полям, особенно когда над ней работает несколько разработчиков.

Методы, свойства и события класса я всегда объявляю закрытыми и невиртуальными. К счастью, C# по умолчанию делает именно так. Разумеется, чтобы типом можно было воспользоваться, некоторые методы, свойства и события должны быть открытыми, но лучше не делать их защищенными или внутренними, поскольку это может сделать тип уязвимым. Впрочем, защищенный или внутренний член все-таки лучше виртуального, поскольку последний предоставляет производному классу большие возможности и всецело зависит от корректности его поведения.

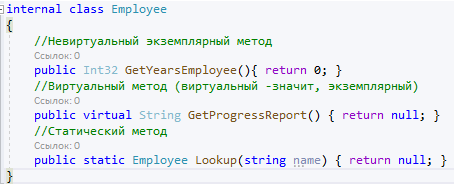
В ООП есть проверенный временем принцип: «лучший метод борьбы со сложностью — добавление новых типов». Если реализация алгоритма чрезмерно усложняется, следует определить вспомогательные типы, инкапсулирующие часть функциональности. Если вспомогательные типы используются в единственном супертипе, следует сделать их вложенными. Это позволит ссылаться на них через супертип и позволит им обращаться к защищенным членам супертипа. Однако существует правило проектирования, примененное в утилите FxCopCmd.exe Visual Studio и рекомендующее определять общедоступные вложенные типы в области видимости файла или сборки (за пределами супертипа), поскольку некоторые разработчики считают синтаксис обращения к вложенным типам громоздким. Я соблюдаю это правило, и никогда не определяю открытые вложенные типы.

# 5 Методы

В этом разделе речь идет только о методах, но все сказанное относится и к виртуальным свойствам и событиям, поскольку они, как показано далее, на самом деле реализуются методами

## 5.1 Вызов виртуальных методов, свойств и событий в CLR

Методы содержат код, выполняющий некоторые действия над типом (статические методы) или экземпляром типа (нестатические). У каждого метода есть имя, сигнатура и возвращаемый тип, который может быть пустым (void). У типа может быть несколько методов с одним именем, но с разным числом параметров или разными возвращаемыми значениями. Можно также определить два метода с одним и тем же именем и параметрами, но с разными типами возвращаемого значения. Благодаря операторам преобразования типов в языке C# это ограничение смягчается (см. главу 8). Определим класс Employee с тремя различными вариантами методов.



При компиляции этого кода компилятор помещает три записи в таблицу определений методов сборки. Каждая запись содержит флаги, указывающие, является ли метод экземплярным, виртуальным или статическим.

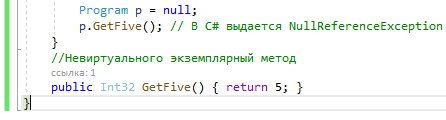
При компиляции кода, ссылающегося на эти методы, компилятор проверяет флаги в определении методов, чтобы выяснить, какой IL-код нужно вставить для корректного вызова методов. В CLR есть две инструкции для вызова метода:

Инструкция call используется для вызова статических, экземплярных и виртуальных методов. Если с помощью этой инструкции вызывается статический метод, необходимо указать тип, в котором определяется метод. При вызове экземплярного или виртуального метода необходимо указать переменную, ссылающуюся на объект, причем в call подразумевается, что эта переменная не равна null. Иначе говоря, сам тип переменной указывает, в каком типе определен необходимый метод. Если в типе переменной метод не определен, проверяются базовые типы. Инструкция call часто служит для невиртуального вызова виртуального метода.

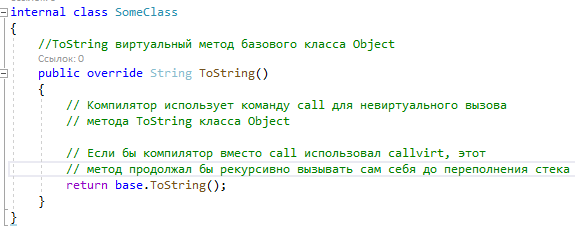
Инструкция callvirt используется только для вызова экземплярных и виртуальных методов. При вызове необходимо указать переменную, ссылающуюся на объект. Если с помощью этой инструкции вызывается невиртуальный экземплярный метод, тип переменной показывает, где определен необходимый метод. При использовании callvirt для вызова виртуального экземплярного метода CLR определяет настоящий тип объекта, на который ссылается переменная, и вызывает метод полиморфно. При компиляции такого вызова JIT-компилятор генерирует код для проверки значения переменной — если оно равно null, CLR сгенерирует исключение NullReferenceException. Из-за этой дополнительной проверки инструкция callvirt выполняется немного медленнее, чем call. Проверка на null выполняется даже при вызове невиртуального экземплярного метода. Давайте посмотрим, как эти инструкции используются в C#.



Поскольку метод WriteLine является статическим, компилятор C# использует для его вызова инструкцию call. Для вызова виртуального метода GetHashCode применяется инструкция callvirt. Наконец, метод GetType также вызывается с помощью инструкции callvirt. Это выглядит странно, поскольку метод GetType невиртуальный. Тем не менее код работает, потому что во время JIT-компиляции CLR знает, что GetType — это невиртуальный метод, и вызывает его невиртуально. Разумеется, возникает вопрос: почему компилятор C# не использует инструкцию call? Разработчики C# решили, что JIT-компилятор должен генерировать код, который проверяет, не равен ли null вызывающий объект. Поэтому вызовы невиртуальных экземплярных методов выполняются чуть медленнее, чем могли бы – а также то, что следующий код в C# вызовет исключение NullReferenceException, хотя в некоторых языках все работает отлично.

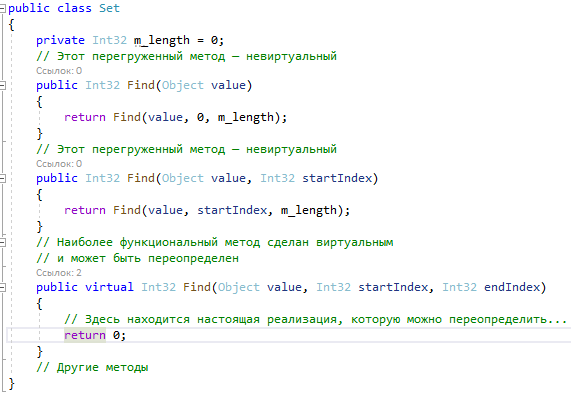


Иногда компилятор вместо callvirt использует для вызова виртуального метода команду call. Такое поведение выглядит странно, но следующий пример показывает, почему это действительно бывает необходимо.



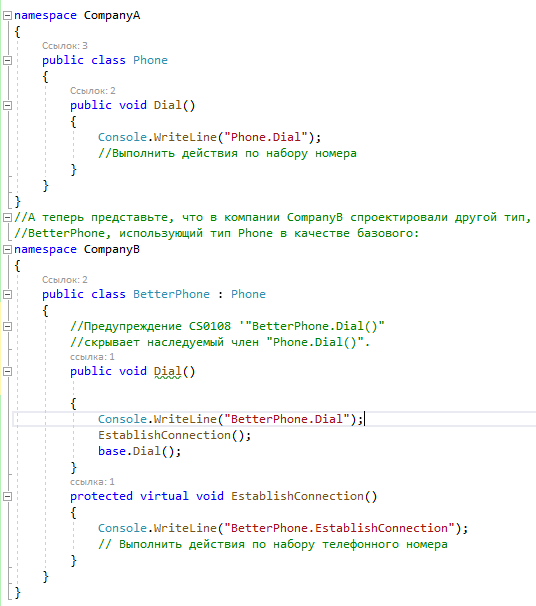
При вызове виртуального метода base.ToString компилятор C# вставляет команду call, чтобы метод ToString базового типа вызывался невиртуально. Это необходимо, ведь если ToString вызвать виртуально, вызов будет выполняться рекурсивно до переполнения стека потока — что, разумеется, нежелательно. Компиляторы стремятся использовать команду call при вызове методов, определенных значимыми типами, поскольку они запечатаны. В этом случае полиморфизм невозможен даже для виртуальных методов, и вызов выполняется быстрее. Кроме того, сама природа экземпляра значимого типа гарантирует, что он никогда не будет равен null, поэтому исключение NullReferenceException не возникнет. Наконец, для виртуального вызова виртуального метода значимого типа CLR необходимо получить ссылку на объект значимого типа, чтобы воспользоваться его таблицей методов, а это требует упаковки значимого типа. Упаковка повышает нагрузку на кучу, увеличивая частоту сборки мусора и снижая производительность.

Независимо от используемой для вызова экземплярного или виртуального метода инструкции — call или callvirt — эти методы всегда в первом параметре получают скрытый аргумент this, ссылающийся на объект, с которым производятся действия. При проектировании типа следует стремиться свести к минимуму количество виртуальных методов. Во-первых, виртуальный метод вызывается медленнее невиртуального. Во-вторых, JIT-компилятор не может подставлять (inline) виртуальные методы, что также ухудшает производительность. В-третьих, как показано далее, виртуальные методы затрудняют управление версиями компонентов. В-четвертых, при определении базового типа часто создается набор перегруженных методов. Чтобы сделать их полиморфными, лучше всего сделать наиболее сложный метод виртуальным, оставив другие методы невиртуальными. Кстати, соблюдение этого правила поможет управлять версиями компонентов, не нарушая работу производных типов. Приведем пример:



## 5.2 Работа с виртуальными методами при управлении версиями типов

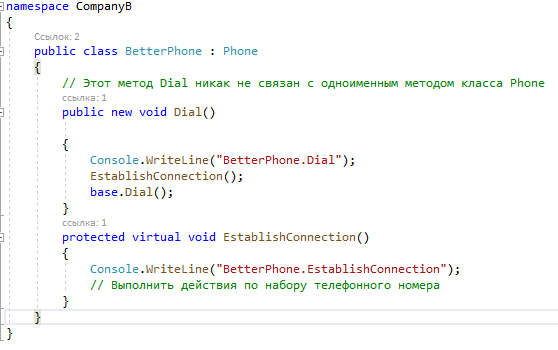
Как уже отмечалось, управление версиями — важный аспект компонентного программирования. Следует быть очень осторожными при добавлении и изменении членов базового типа. Рассмотрим несколько примеров. Пусть разработчиками компании CompanyА спроектирован тип Phone:



При попытке скомпилировать свой код разработчики компании CompanyB получают от компилятора C# предупреждение:



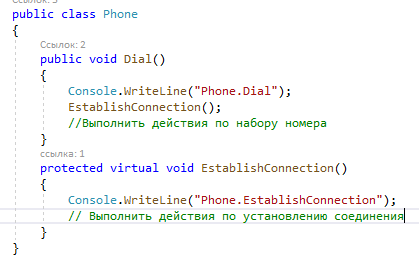
Смысл в том, что метод Dial, определяемый в типе BetterPhone, скроет одноименный метод в Phone. В новой версии метода Dial его семантика может стать совсем иной, нежели та, что определена программистами компании CompanyА в исходной версии метода. Предупреждение о таких потенциальных семантических несоответствиях — очень полезная функция компилятора. Компилятор также подсказывает, как избавиться от этого предупреждения: нужно поставить ключевое слово new перед определением метода Dial в классе BetterPhone. Вот как выглядит исправленный класс BetterPhone:



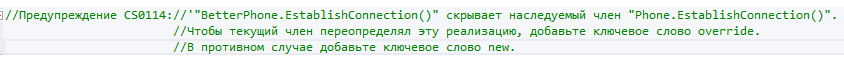
Теперь компания CompanyB может использовать в своем приложении тип BetterPhone следующим образом:



Результат свидетельствует о том, что код выполняет именно те действия, которые нужны компании CompanyB. При вызове Dial вызывается новая версия этого метода, определенная в типе BetterPhone. Она сначала вызывает виртуальный метод EstablishConnection, а затем — исходную версию метода Dial из базового типа Phone. А теперь представим, что несколько компаний решили использовать тип Phone, созданный в компании CompanyА. Допустим также, что все они сочли полезным установление соединения в самом методе Dial. Эти отзывы заставили разработчиков компании CompanyА усовершенствовать класс Phone:

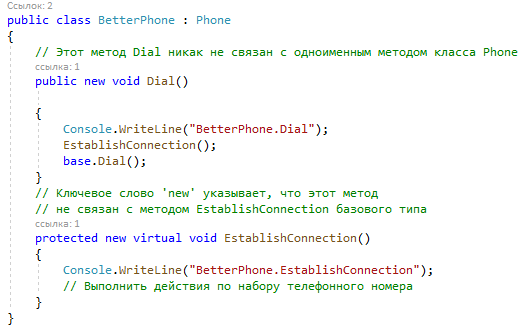


В результате теперь разработчики компании CompanyB при компиляции своего типа BetterPhone (производного от новой версии Phone) получают следующее предупреждение:



В нем говорится о том, что 'BetterPhone. EstablishConnection()' скрывает унаследованный член 'Phone.EstablishConnection()', и чтобы текущий член переопределил реализацию, нужно вставить ключевое слово override; в противном случае нужно вставить ключевое слово new. То есть компилятор предупреждает, что как Phone, так и BetterPhone предлагают метод EstablishConnection, семантика которого может отличаться в разных классах. В этом случае простая перекомпиляция BetterPhone больше не может гарантировать, что новая версия метода будет работать так же, как прежняя, определенная в типе Phone.

Если в компании CompanyB решат, что семантика метода EstablishConnection в этих двух типах отличается, компилятору будет указано, что «правильными» являются методы Dial и EstablishConnection, определенные в BetterPhone, и они не связаны с одноименными методами из базового типа Phone. Для этого разработчики компании CompanyB добавляют ключевое слово new в определение EstablishConnection:



Здесь ключевое слово new заставляет компилятор сгенерировать метаданные, информирующие CLR, что определенные в BetterPhone методы Dial и EstablishConnection следует рассматривать как новые функции, введенные в этом типе. При этом CLR будет известно, что одноименные методы типов Phone и BetterPhone никак не связаны.

При выполнении того же приложения (метода Main) выводится информация:

BetterPhone.Dial

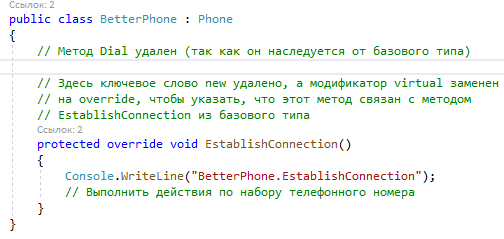
BetterPhone.EstablishConnection

Phone.Dial

Phone.EstablishConnection.

Отсюда видно, что, когда Main обращается к методу Dial, вызывается версия, определенная в BetterPhone. Далее Dial вызывает виртуальный метод EstablishConnection, также определенный в BetterPhone. Когда метод EstablishConnection типа BetterPhone возвращает управление, вызывается метод Dial типа Phone, вызывающий метод EstablishConnection этого типа. Но поскольку метод EstablishConnection в типе BetterPhone помечен ключевым словом new, вызов этого метода не считается переопределением виртуального метода EstablishConnection, исходно определенного в типе Phone. В результате метод Dial типа Phone вызывает метод EstablishConnection, определенный в типе Phone, что и требовалось от программы.

Альтернативное решение: CompanyB, получив от CompanyА новую версию типа Phone, решает, что текущая семантика методов Dial и EstablishConnection типа Phone — это именно то, что нужно. В этом случае в CompanyB полностью удаляют метод Dial из типа BetterPhone. Поскольку теперь разработчикам CompanyB нужно указать компилятору, что метод EstablishConnection типа BetterPhone связан с одноименным методом типа Phone, нужно удалить из его определения ключевое слово new. Удаления ключевого слова недостаточно, так как компилятор не поймет предназначения метода EstablishConnection типа BetterPhone. Чтобы выразить намерения явно, разработчик из CompanyB должен изменить модификатор определенного в типе BetterPhone метода EstablishConnection с virtual на override. Код новой версии BetterPhone выглядит так:



Теперь то же приложение (метод Main) выводит следующий результат:

Phone.Dial

BetterPhone.EstablishConnection.

Видно, что когда Main вызывает метод Dial, вызывается версия этого метода, определенная в типе Phone и унаследованная от него типом BetterPhone. Далее, когда метод Dial, определенный в типе Phone, вызывает виртуальный метод EstablishConnection, вызывается одноименный метод типа BetterPhone, так как он переопределяет виртуальный метод EstablishConnection, определяемый типом Phone.

## 5.3 Конструкторы экземпляров и классы (ссылочные типы)

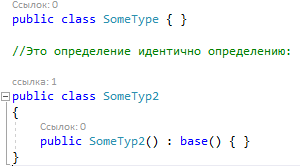
Конструкторы — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа. В таблице определений, входящих в метаданные, методы-конструкторы всегда отмечают сочетанием .ctor (от constructor). При создании экземпляра объекта ссылочного типа выделяется память для полей данных экземпляра и инициализируются служебные поля (указатель на объект-тип и индекс блока синхронизации), после чего вызывается конструктор экземпляра, устанавливающий исходное состояние нового объекта.

При конструировании объекта ссылочного типа выделяемая для него память всегда обнуляется до вызова конструктора экземпляра типа. Любые поля, не задаваемые конструктором явно, гарантированно содержат 0 или null.

В отличие от других методов конструкторы экземпляров не наследуются. Иначе говоря, у класса есть только те конструкторы экземпляров, которые определены в этом классе. Невозможность наследования означает, что к конструктору экземпляров нельзя применять модификаторы virtual, new, override, sealed и abstract.

### 5.3.1 Конструктор по умолчанию (без параметров)

Если определить класс без явно заданных конструкторов, многие компиляторы (в том числе компилятор C#) создадут конструктор по умолчанию (без параметров), реализация которого просто вызывает конструктор без параметров базового класса. Например, рассмотрим следующее определение класса:



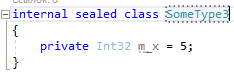
Для абстрактных классов компилятор создает конструктор по умолчанию с модификатором protected, в противном случае область действия будет открытой (public). Если в базовом классе нет конструктора без параметров, производный класс должен явно вызвать конструктор базового класса, иначе компилятор вернет ошибку. Для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию.

В типе может определяться несколько конструкторов, при этом сигнатуры и уровни доступа к конструкторам обязательно должны отличаться. В случае верифицируемого кода конструктор экземпляров должен вызывать конструктор базового класса до обращения к какому-либо из унаследованных от него полей. Многие компиляторы, включая C#, генерируют вызов конструктора базового класса автоматически. В конечном счете всегда вызывается открытый конструктор объекта System.Object без параметров. Этот конструктор ничего не делает — просто возвращает управление по той простой причине, что в System.Object не определено никаких экземплярных полей данных, поэтому конструктору просто нечего делать.

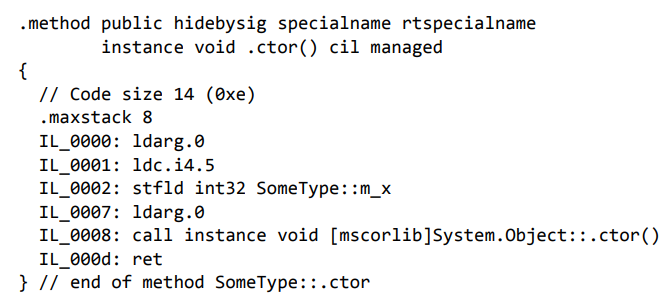
В редких ситуациях экземпляр типа может создаваться без вызова конструктора экземпляров. В частности, метод MemberwiseClone объекта Object выделяет память, инициализирует служебные поля объекта, а затем копирует байты исходного объекта в область памяти, выделенную для нового объекта. Кроме того, конструктор обычно не вызывается при десериализации объекта. Код десериализации выделяет память для объекта без вызова конструктора, используя метод GetUninitializedObject или GetSafeUninitializedObject типа System.Runtime. Serialization.FormatterServices (см. главу 24).

### 5.3.2 Создание конструктор для ссылочных типов (class)

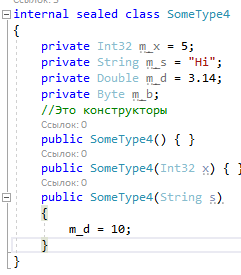
C# предлагает простой синтаксис, позволяющий инициализировать поля во время создания объекта ссылочного типа:



При создании объекта SomeType его поле m\_x инициализируется значением 5. Вы можете спросить: как это происходит? Изучив IL-код метода-конструктора этого объекта (этот метод также фигурирует под именем .ctor), вы увидите следующий код:

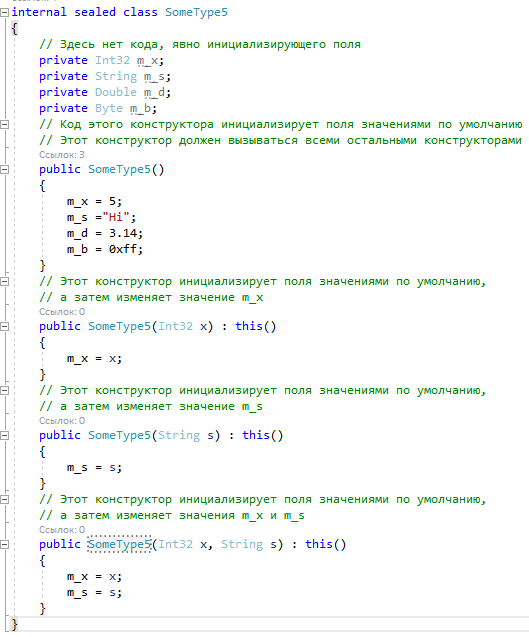


Как видите, конструктор объекта SomeType содержит код, записывающий в поле m\_x значение 5 и вызывающий конструктор базового класса. Иначе говоря, компилятор C# предлагает удобный синтаксис, позволяющий инициализировать поля экземпляра при их объявлении. Компилятор транслирует этот синтаксис в методконструктор, выполняющий инициализацию. Это значит, что нужно быть готовым к разрастанию кода, как это показано на следующем примере:



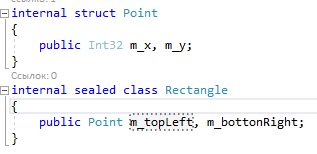
Генерируя IL-код для трех методов-конструкторов из этого примера, компилятор помещает в начало каждого из методов код, инициализирующий поля m\_x, m\_s и m\_d. После кода инициализации вставляется вызов конструктора базового класса, а затем добавляется код, расположенный внутри методов-конструкторов. Например, IL-код, сгенерированный для конструктора с параметром типа String, состоит из кода, инициализирующего поля m\_x, m\_s и m\_d, и кода, перезаписывающего поле m\_d значением 10. Заметьте: поле m\_b гарантированно инициализируется значением 0, даже если нет кода, инициализирующего это поле явно.

Поскольку в показанном ранее классе определены три конструктора, компилятор трижды генерирует код, инициализирующий поля m\_x, m\_s и m\_d: по одному разу для каждого из конструкторов. Если имеется несколько инициализируемых экземплярных полей и множество перегруженных методов-конструкторов, стоит подумать о том, чтобы определить поля без инициализации; создать единственный конструктор, выполняющий общую инициализацию, и заставить каждый методконструктор явно вызывать конструктор, выполняющий общую инициализацию. Этот подход позволит уменьшить размер генерируемого кода. Следующий пример иллюстрирует использование способности C# явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова this:

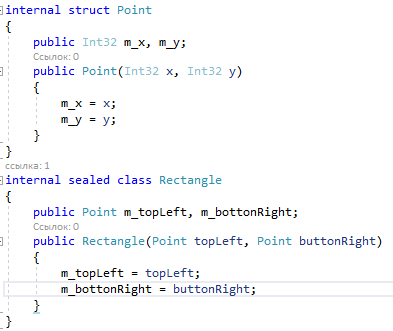


### 5.3.3 Конструкторы значимых типов (struct)

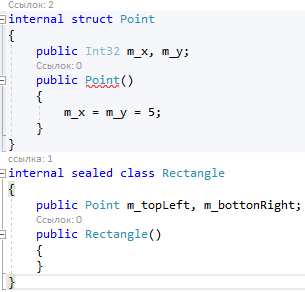
Конструкторы значимых типов (struct) работают иначе, чем ссылочных (class). CLR всегда разрешает создание экземпляров значимых типов и этому ничто не может помешать. Поэтому, по большому счету, конструкторы у значимого типа можно не определять. Фактически многие компиляторы (включая C#) не определяют для значимых типов конструкторы по умолчанию, не имеющие параметров. Разберем следующий код:



Для того чтобы создать объект Rectangle, надо использовать оператор new с указанием конструктора. В этом случае вызывается конструктор, автоматически сгенерированный компилятором C#. Память, выделенная для объекта Rectangle, включает место для двух экземпляров значимого типа Point. Из соображений повышения производительности CLR не пытается вызвать конструктор для каждого экземпляра значимого типа, содержащегося в объекте ссылочного типа. Однако, как отмечалось ранее, поля значимого типа инициализируются нулями/null. Вообще говоря, CLR позволяет программистам определять конструкторы для значимых типов, но эти конструкторы выполняются лишь при наличии кода, явно вызывающего один из них, например, как в конструкторе объекта Rectangle:



Конструктор экземпляра значимого типа выполняется только при явном вызове. Так что если конструктор объекта Rectangle не инициализировал его поля m\_topLeft и m\_bottomRight вызовом с помощью оператора new конструктора Point, поля m\_x и m\_y у обеих структур Point будут содержать 0. Если значимый тип Point уже определен, то определяется конструктор, по умолчанию не имеющий параметров. Однако давайте перепишем наш код:



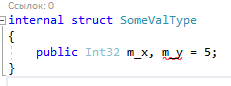
А теперь ответьте, какими значениями — 0 или 5 — будут инициализированы поля m\_x и m\_y, принадлежащие структурам Point (m\_topLeft и m\_bottomRight)? Предупреждаю, вопрос с подвохом.

Подвох в том, что C# не позволяет определять для значимого типа конструкторы без параметров. Поэтому показанный код на самом деле даже не компилируется. При попытке скомпилировать его компилятор C# генерирует сообщение об ошибке (ошибка CS0568: структура не может содержать явные конструкторы без параметров)

В отсутствие конструктора без параметров поля значимого типа всегда инициализируются нулями/null.

Поскольку C# не допускает использования значимых типов с конструкторами без параметров, при компиляции следующего типа компилятор сообщает об ошибке: (ошибка CS0573: 'SomeValType.m\_x': нельзя создавать инициализаторы экземплярных полей в структурах)

А вот как выглядит код, вызвавший эту ошибку:

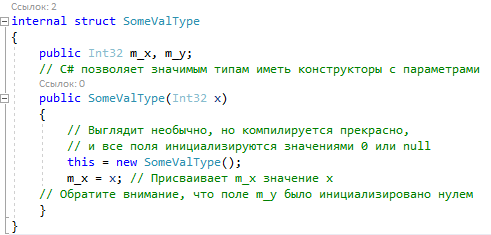


Кроме того, поскольку верифицируемый код перед чтением любого поля значимого типа требует записывать в него какое-либо значение, любой конструктор, определенный для значимого типа, должен инициализировать все поля этого типа. Следующий тип определяет конструктор для значимого типа, но не может инициализировать все его поля:



При компиляции этого типа компилятор C# генерирует сообщение об ошибке: (ошибка CS0171: поле 'SomeValType.m\_y' должно быть полностью определено до возвращения управления конструктором)

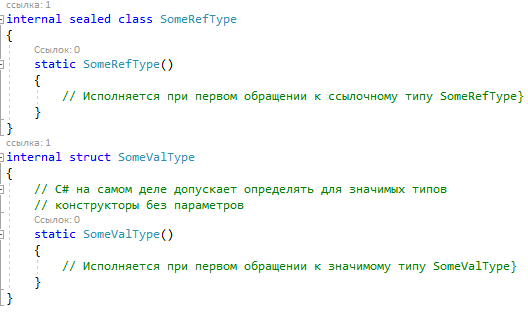
Чтобы разрешить проблему, конструктор должен ввести в поле y какое-нибудь значение (обычно 0). В качестве альтернативного варианта можно инициализировать все поля значимого типа, как это сделано здесь:



В конструкторе значимого типа this представляет экземпляр значимого типа и ему можно приписать значение нового экземпляра значимого типа, у которого все поля инициализированы нулями. В конструкторах ссылочного типа указатель this считается доступным только для чтения и присваивать ему значение нельзя.

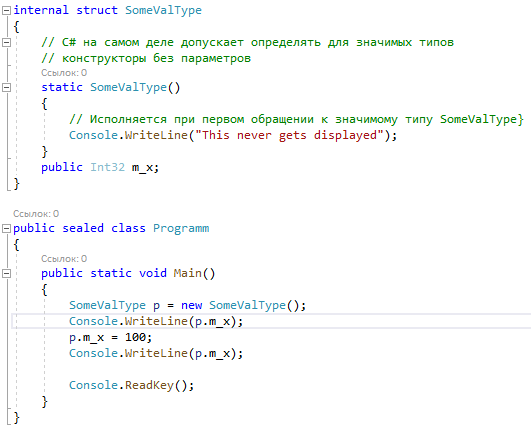
### 5.3.4 Конструкторы типов (статические конструкторы, конструкторы классов и инициализаторы типов)

Помимо конструкторов экземпляров, CLR поддерживает конструкторы типов (также известные как статические конструкторы, конструкторы классов и инициализаторы типов). Конструкторы типов можно применять и к интерфейсам (хотя C# этого не допускает), ссылочным и значимым типам. Подобно тому, как конструкторы экземпляров используются для установки первоначального состояния экземпляра типа, конструкторы типов служат для установки первоначального состояния типа. По умолчанию у типа не определено конструктора. У типа не может быть более одного конструктора; кроме того, у конструкторов типов никогда не бывает параметров. Вот как определяются ссылочные и значимые типы с конструкторами в программах на C#:

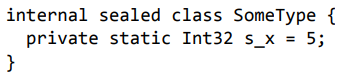


Обратите внимание, что конструкторы типов определяют так же, как конструкторы экземпляров без параметров за исключением того, что их помечают как статические. Кроме того, конструкторы типов всегда должны быть закрытыми (C# делает их закрытыми автоматически). Однако если явно пометить в исходном тексте программы конструктор типа как закрытый (или как-то иначе), компилятор C# выведет сообщение об ошибке: (ошибка CS0515: 'SomeValType.Some-ValType()': в статических конструкторах нельзя использовать модификаторы доступа): error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on static constructors Конструкторы типов всегда должны быть закрытыми, чтобы код разработчика не смог их вызвать, напротив, в то же время среда CLR всегда способна вызвать конструктор типа.

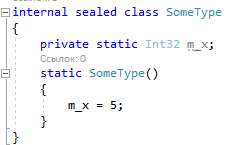
Конструкторы типов всегда должны быть закрытыми, чтобы код разработчика не смог их вызвать, напротив, в то же время среда CLR всегда способна вызвать конструктор типа.

Внимание Хотя конструктор типа можно определить в значимом типе, этого никогда не следует делать, так как иногда CLR не вызывает статический конструктор значимого типа:

Код конструктора типа может обращаться только к статическим полям типа; обычно это делается, чтобы их инициализировать. Как и в случае экземплярных полей, C# предлагает простой синтаксис:

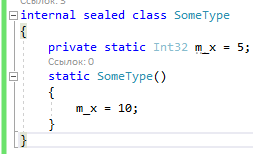


При компоновке этого кода компилятор автоматически генерирует конструктор типа SomeType. Иначе говоря, получается тот же эффект, как если бы этот код был написан следующим образом:



Конструктор типа не должен вызывать конструктор базового класса. Этот вызов не обязателен, так как ни одно статическое поле типа не используется совместно с базовым типом и не наследуется от него.

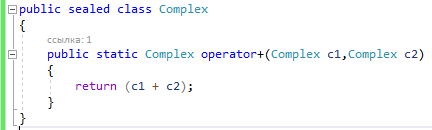
В завершение этого раздела рассмотрим следующий код:



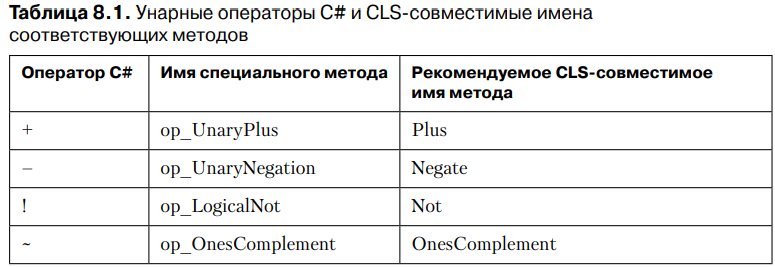
Здесь компилятор C# генерирует единственный метод-конструктор типа, который сначала инициализирует поле s\_x значением 5, затем — значением 10. Иначе говоря, при генерации IL-кода конструктора типа компилятор C# сначала генерирует код, инициализирующий статические поля, затем обрабатывает явный код, содержащийся внутри метода-конструктора типа.

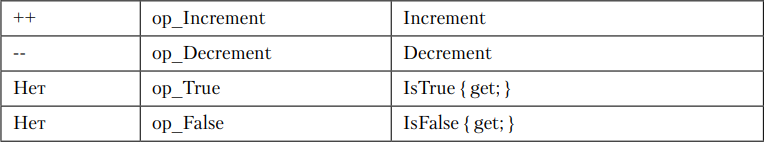
## 5.4 Методы перегруженных операторов

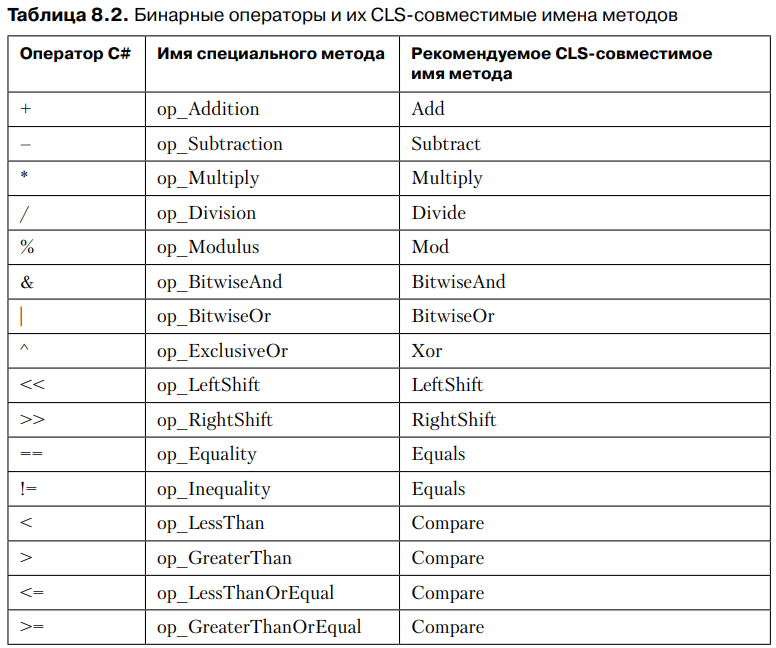
В некоторых языках тип может определять, как операторы должны манипулировать его экземплярами. В частности, многие типы (например, System.String, System. Decimal и System.DateTime) используют перегрузку операторов равенства (==) и неравенства (!=). CLR ничего не известно о перегрузке операторов — ведь среда даже не знает, что такое оператор. Смысл операторов и код, который должен быть сгенерирован, когда тот или иной оператор встретится в исходном тексте, определяется языком программирования. Например, если в программе на C# поставить между обычными числами оператор +, компилятор генерирует код, выполняющий сложение двух чисел. Когда оператор + применяют к строкам, компилятор C# генерирует код, выполняющий конкатенацию этих строк. Для обозначения неравенства в C# используется оператор !=,. Наконец, оператор ^ в C# задает операцию «исключающее или» (XOR). Хотя CLR ничего не знает об операторах, среда указывает, как языки программирования должны предоставлять доступ к перегруженным операторам, чтобы последние могли легко использоваться в коде на разных языках программирования. Для каждого конкретного языка проектировщики решают, будет ли этот язык поддерживать перегрузку операторов и, если да, какой синтаксис задействовать для представления и использования перегруженных операторов. С точки зрения CLR перегруженные операторы представляют собой просто методы. От выбора языка зависит наличие поддержки перегруженных операторов и их синтаксис, а при компиляции исходного текста компилятор генерирует метод, определяющий работу оператора. Спецификация CLR требует, чтобы перегруженные операторные методы были открытыми и статическими. Дополнительно C# (и многие другие языки) требует, чтобы у операторного метода тип, по крайней мере, одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод. Причина этого ограничения в том, что оно позволяет компилятору C# в разумное время находить кандидатуры операторных методов для привязки. Пример метода перегруженного оператора, заданного в определении класса C#:



Компилятор генерирует определение метода op\_Addition и устанавливает в записи с определением этого метода флаг specialname, свидетельствующий о том, что это «особый» метод. Когда компилятор языка (в том числе компилятор C#) видит в исходном тексте оператор +, он исследует типы его операндов. При этом компилятор пытается выяснить, не определен ли для одного из них метод op\_Addition с флагом specialname, параметры которого совместимы с типами операндов. Если такой метод существует, компилятор генерирует код, вызывающий этот метод, иначе возникает ошибка компиляции. В табл. 8.1 и 8.2 приведен набор унарных и бинарных операторов, которые C# позволяет перегружать, их обозначения и рекомендованные имена соответствующих методов, которые должен генерировать компилятор. Третий столбец я прокомментирую в следующем разделе.

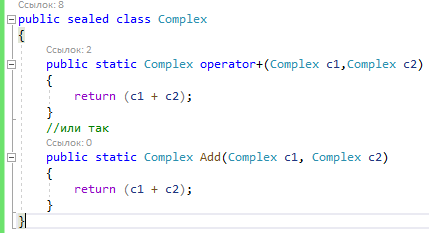




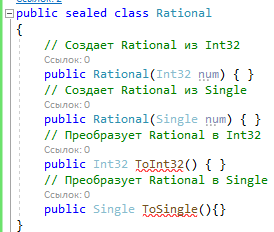


В спецификации CLR определены многие другие операторы, поддающиеся перегрузке, но C# их не поддерживает. Они не очень распространены, поэтому я их здесь не указал. Полный список есть в спецификации ECMA (www.ecma-international.org/ publications/standards/Ecma-335.htm) общеязыковой инфраструктуры CLI, разделы 10.3.1 (унарные операторы) и 10.3.2 (бинарные операторы).

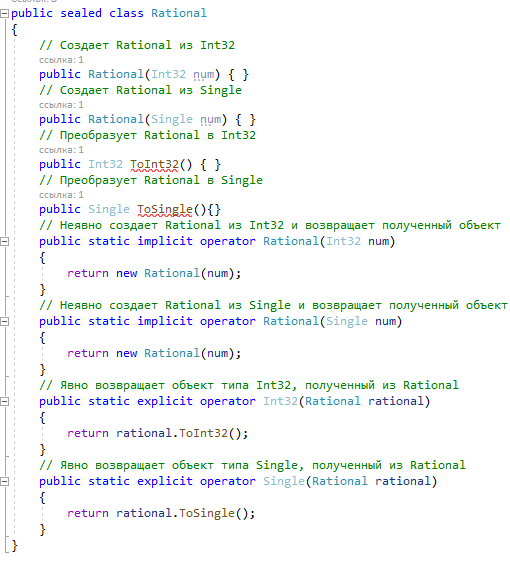
Для типа с методами перегруженных операторов Microsoft также рекомендует определять открытые экземплярные методы с дружественными именами, вызывающие методы перегруженных операторов в своей внутренней реализации. Например, тип с перегруженными методами op\_Addition или op\_AdditionAssignment должен также определять открытый метод с дружественным именем Add. Список рекомендованных дружественных имен для всех методов операторов приводится в третьем столбце табл. 8.1 и 8.2. Таким образом, показанный ранее тип Complex можно было бы определить и так:



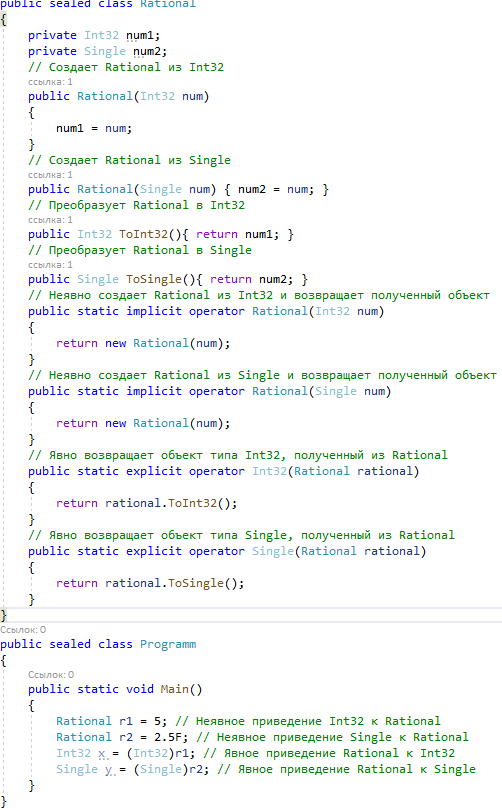
## 5.5 Методы операторов преобразования

Время от времени возникает необходимость в преобразовании объекта одного типа в объект другого типа. Уверен, что вам приходилось преобразовывать значение Byte в Int32. Когда исходный и целевой типы являются примитивными, компилятор способен без посторонней помощи генерировать код, необходимый для преобразования объекта. Если ни один из типов не является примитивным, компилятор генерирует код, заставляющий CLR выполнить преобразование (приведение типов). В этом случае CLR просто проверяет, совпадает ли тип исходного объекта с целевым типом (или является производным от целевого). Однако иногда требуется преобразовать объект одного типа в совершенно другой тип. Например, класс System.Xml.Linq. XElement позволяет преобразовать элемент XML в Boolean, (U)Int32, (U)Int64, Single, Double, Decimal, String, DateTime, DateTimeOffset, TimeSpan, Guid или эквивалент любого из этих типов, допускающий присваивание null (кроме String). Также можно представить, что в FCL есть тип данных Rational, в который удобно преобразовывать объекты типа Int32 или Single. Более того, было бы полезно иметь возможность выполнить обратное преобразование объекта Rational в Int32 или Single. Для выполнения этих преобразований в типе Rational должны определяться открытые конструкторы, принимающие в качестве единственного параметра экземпляр преобразуемого типа. Кроме того, нужно определить открытый экземплярный метод ToXxx, не принимающий параметров (как популярный метод ToString). Каждый такой метод преобразует экземпляр типа, в котором определен этот метод, в экземпляр типа Xxx. Вот как правильно определить соответствующие конструкторы и методы для типа Rational:

Вызывая эти конструкторы и методы, разработчик, используя любой язык, может преобразовать объект типа Int32 или Single в Rational и обратно. Подобные преобразования могут быть весьма удобны, и при проектировании типа стоит подумать, какие конструкторы и методы преобразования имело бы смысл включить в него. Ранее мы обсуждали способы поддержки перегрузки операторов в разных языках. Некоторые (например, C#) наряду с этим поддерживают перегрузку операторов преобразования — методы, преобразующие объекты одного типа в объекты другого типа. Методы операторов преобразования определяются при помощи специального синтаксиса. Спецификация CLR требует, чтобы перегруженные методы преобразования были открытыми и статическими. Кроме того, C# (и многие другие языки) требуют, чтобы у метода преобразования тип, по крайней мере, одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод. Причина этого ограничения в том, что оно позволяет компилятору C# в разумное время находить кандидатуры операторных методов для привязки. Следующий код добавляет в тип Rational четыре метода операторов преобразования:



При определении методов для операторов преобразования следует указать, должен ли компилятор генерировать код для их неявного вызова автоматически или лишь при наличии явного указания в исходном тексте. Ключевое слово implicit указывает компилятору C#, что наличие в исходном тексте явного приведения типов не обязательно для генерации кода, вызывающего метод оператора преобразования. Ключевое слово explicit позволяет компилятору вызывать метод только тогда, когда в исходном тексте происходит явное приведение типов. После ключевого слова implicit или explicit вы сообщаете компилятору, что данный метод представляет собой оператор преобразования (ключевое слово operator). После ключевого слова operator указывается целевой тип, в который преобразуется объект, а в скобках — исходный тип объекта. Определив в показанном ранее типе Rational операторы преобразования, можно написать (на C#):



Компилятор C# полностью поддерживает операторы преобразования. Обнаружив код, в котором вместо ожидаемого типа используется объект совсем другого типа, компилятор ищет метод оператора неявного преобразования, способный выполнить нужное преобразование, и генерирует код, вызывающий этот метод. Если подходящий метод оператора неявного преобразования обнаруживается, компилятор вставляет в результирующий IL-код вызов этого метода. Найдя в исходном тексте явное приведение типов, компилятор ищет метод оператора явного или неявного преобразования. Если он существует, компилятор генерирует вызывающий его код. Если компилятор не может найти подходящий метод оператора преобразования, он выдает ошибку, и код не компилируется.

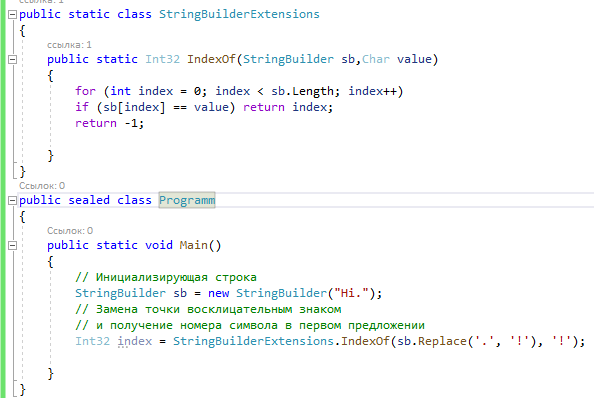
ПримечАние

С# генерирует код вызова операторов неявного преобразования в случае, когда используется выражение приведения типов. Однако операторы неявного преобразования никогда не вызываются, если используется оператор as или is.

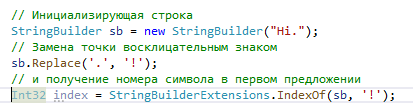
Чтобы по-настоящему разобраться в методах перегруженных операторов и операторов преобразования, я настоятельно рекомендую использовать тип System. Decimal как образец. В типе Decimal определено несколько конструкторов, позволяющих преобразовывать в Decimal объекты различных типов. Он также поддерживает несколько методов ToXxx для преобразования объектов типа Decimal в объекты других типов. Наконец, в этом типе определен ряд методов операторов преобразования и перегруженных операторов.

## 5.6 Методы расширения

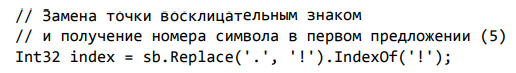
Механизм методов расширения лучше всего рассматривать на конкретном примере. В главе 14 я упоминаю о том, что для управления строками класс StringBuilder предлагает меньше методов, чем класс String, и это довольно странно, потому что класс StringBuilder является предпочтительнее для управления строками, так как он изменяем. Допустим, вы хотите определить некоторые отсутствующие в классе StringBuilder методы самостоятельно. Возможно, вы решите определить собственный метод IndexOf:



Этот программный код работает, но в перспективе он не идеален. Во-первых, программист, желающий получить индекс символа при помощи класса StringBuilder, должен знать о существовании класса StringBuilderExtensions. Во-вторых, программный код не отражает последовательность операторов, представленных в объекте StringBuilder, что усложняет понимание, чтение и сопровождение кода. Программистам удобнее было бы вызывать сначала метод Replace, а затем метод IndexOf, но когда вы прочитаете последнюю строчку кода слева направо, первым в строке окажется IndexOf, а затем — Replace. Вы можете исправить ситуацию и сделать поведение программного кода более понятным, написав следующий код:

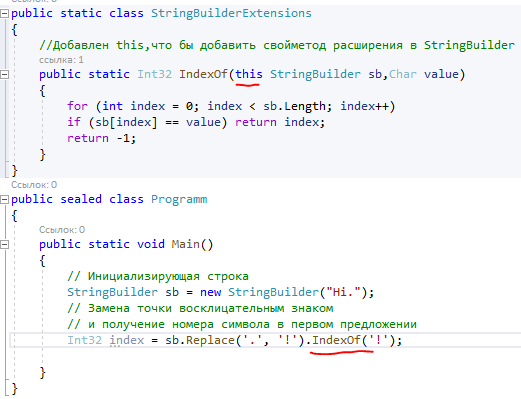


Однако здесь возникает третья проблема, затрудняющая понимание логики кода. Использование класса StringBuilderExtensions отвлекает программиста от выполняемой операции: IndexOf. Если бы класс StringBuilder определял собственный метод IndexOf, то представленный код можно было бы переписать следующим образом:



В контексте сопровождения программного кода это выглядит великолепно! В объекте StringBuilder мы заменяем точку восклицательным знаком, а затем находим индекс этого знака.

А сейчас я попробую объяснить, что именно делают методы расширения. Они позволяют вам определить статический метод, который вызывается посредством синтаксиса экземплярного метода. Иначе говоря, мы можем определить собственный метод IndexOf — и три проблемы, упомянутые выше, исчезнут. Для того чтобы превратить метод IndexOf в метод расширения, мы просто добавим ключевое слово this перед первым аргументом:



Компилятор увидит следующий код:

Int32 index = sb.IndexOf('X');

Сначала он проверит класс StringBuilder или все его базовые классы, предоставляющие экземплярные методы с именем IndexOf и единственным параметром Char. Если они не существуют, тогда компилятор будет искать любой статический класс с определенным методом IndexOf, у которого первый параметр соответствует типу выражения, используемого при вызове метода. Этот тип должен быть отмечен при помощи ключевого слова this. В данном примере выражением является sb типа StringBuilder. В этом случае компилятор ищет метод IndexOf с двумя параметрами: StringBuilder (отмеченное словом this) и Char. Компилятор найдет наш метод IndexOf и сгенерирует IL-код для вызова нашего статического метода. Теперь понятно, как компилятор решает две последние упомянутые мной проблемы, относящиеся к читабельности кода. Однако до сих пор непонятно, как решается первая проблема, то есть как программисты узнают о том, что метод IndexOf существует и может использоваться в объекте StringBuilder? Ответ на этот вопрос в Microsoft Visual Studio дает механизм IntelliSense. В редакторе, когда вы напечатаете точку, появится IntelliSense-окно со списком доступных методов. Кроме того, в IntelliSense-окне будут представлены все методы расширения, существующие для типа выражения, написанного слева от точки. IntelliSense-окно показано на рис. 8.1. Как видите, рядом с методами расширения имеется стрелочка, а контекстная подсказка показывает, что метод действительно является методом расширения. Это очень удобно, потому что теперь при помощи этого инструмента вы можете легко определять собственные методы для управления различными типами объектов, а другие программисты естественным образом узнают о них при использовании объектов этих типов.

### 5.6.1 Правила и рекомендации

Приведу несколько правил и фактов, которые необходимо знать о методах расширения.

Язык С# поддерживает только методы расширения, он не поддерживает свойств расширения, событий расширения, операторов расширения и т. д.

Методы расширения (методы со словом this перед первым аргументом) должны быть объявлены в статическом необобщенном классе. Однако нет ограничения на имя этого класса, вы можете назвать его как вам угодно. Конечно, метод расширения должен иметь, по крайней мере, один параметр, и только первый параметр может быть отмечен ключевым словом this.

Компилятор C# ищет методы расширения, заданные только в статических классах, определенных в области видимости файла. Другими словами, если вы определили статический класс, унаследованный от другого класса, компилятор C# выдаст следующее сообщение (ошибка CS1109: метод расширения должен быть определен в статическом классе первого уровня, StringBuilderExtensions является вложенным классом): error CS1109: Extension method must be defined in a top-level static class; StringBuilderExtensions is a nested class

Так как статическим классам можно давать любые имена по вашему желанию, компилятору С# необходимо какое-то время для того, чтобы найти методы расширения; он просматривает все статические классы, определенные в области файла, и сканирует их статические методы. Для повышения производительности и для того, чтобы не рассматривать лишние в данных обстоятельствах методы расширения, компилятор C# требует «импортирования» методов расширения. Например, пусть кто-нибудь определил класс StringBuilderExtensions в пространстве имен Wintellect, тогда другой программист, которому нужно иметь доступ к методу расширения данного класса, в начале файла программного кода должен указать команду using Wintellect.

Существует возможность определения в нескольких статических классах одинаковых методов расширения. Если компилятор выяснит, что существуют два и более методов расширения, то тогда он выдает следующее сообщение (ошибка CS0121: неоднозначный вызов следующих методов или свойств 'StringBuilderExtensions.IndexOf(string, char)' и 'AnotherStringBuild erExtensions.IndexOf(string, char)):

error CS0121: The call is ambiguous between the following methods or properties: 'StringBuilderExtensions.IndexOf(string, char)' and 'AnotherStringBuilderExtensions.IndexOf(string, char)'.

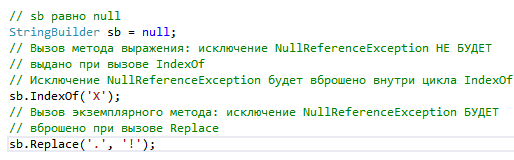
Для того чтобы исправить эту ошибку, вы должны модифицировать программный код. Нельзя использовать синтаксис экземплярного метода для вызова статического метода, вместо этого должен применяться синтаксис статического метода с указанием имени статического класса, чтобы точно сообщить компилятору, какой именно метод нужно вызвать.

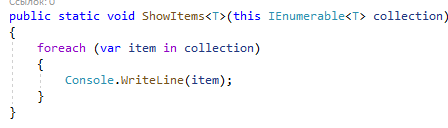
Прибегать к этому механизму следует не слишком часто, так как он известен не всем разработчикам. Например, когда вы расширяете тип с методом расширения, вы действительно расширяете унаследованные типы с этим методом. Следовательно, вы не должны определять метод выражения, чей первый параметр — System.Object, так как этот метод будет вызываться для всех типов выражений, и соответствующие ссылки только будут загромождать окно IntelliSense.

Существует потенциальная проблема с версиями. Если в будущем разработчики Microsoft добавят экземплярный метод IndexOf к классу StringBuilder с тем же прототипом, что и в моем примере, то когда я перекомпилирую свой программный код, компилятор свяжет с программой экземплярный метод IndexOf компании Microsoft вместо моего статического метода IndexOf. Из-за этого моя программа начнет себя по-другому. Эта проблема версий — еще одна причина, по которой этот механизм следует использовать осмотрительно.

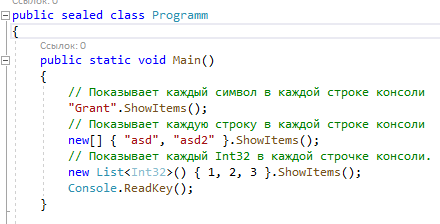
### 5.6.2 Расширение разных типов методами расширения

В этой главе я продемонстрировал, как определять методы расширения для класса StringBuilder. Я хотел бы отметить, что так как метод расширения на самом деле является вызовом статического метода, то среда CLR не генерирует код для проверки значения выражения, используемого для вызова метода (равно ли оно null).

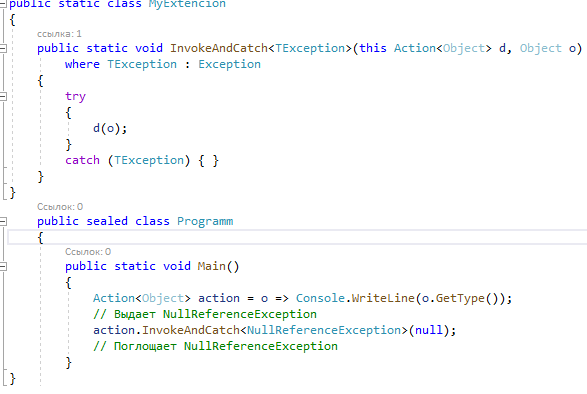


Я также хотел бы отметить, что вы можете определять методы расширения для интерфейсных типов, как в следующем программном коде:

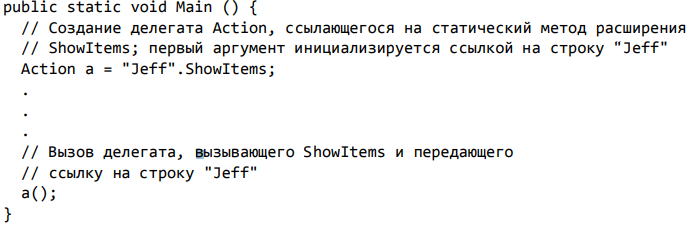
Представленный здесь метод расширения может быть вызван с использованием любого выражения, результат выполнения которого относится к типу, реализующему интерфейс IEnumerable:



Методы расширения также можно определять и для типов-делегатов, например:



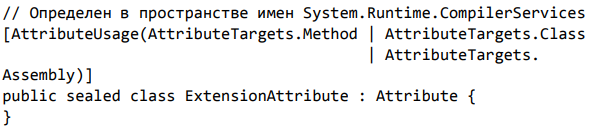
Кроме того, можно добавлять методы расширения к перечислимым типам (примеры см. в главе 15).

Наконец, компилятор С# позволяет создавать делегатов, ссылающихся на метод расширения через объект (см. главу 17):

В представленном программном коде компилятор С# генерирует IL-код для того, чтобы создать делегата Action. После создания делегата конструктор передается в вызываемый метод, также передается ссылка на объект, который должен быть передан в этот метод в качестве скрытого параметра. Обычно, когда вы создаете делегата, ссылающегося на статический метод, объектная ссылка равна null, потому что статический метод не имеет этого параметра. Однако в данном примере компилятор C# сгенерирует специальный код, создающий делегата, ссылающегося на статический метод ShowItems, а целевым объектом статического метода будет ссылка на строку "Jeff". Позднее, при вызове делегата, CLR вызовет статический метод и передаст ему ссылку на строку "Jeff". Все это напоминает какие-то фокусы, но хорошо работает и выглядит естественно, если не думать, что при этом происходит внутри.

### 5.6.3 Атрибут расширения

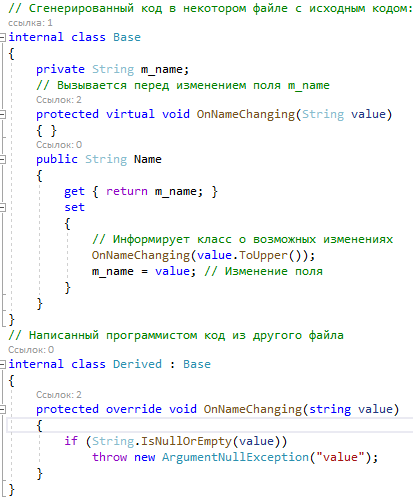
Конечно, было бы лучше, чтобы концепция методов расширения относилась бы не только к С#. Хотелось бы, чтобы программисты определяли набор методов расширения на разных языках программирования и, таким образом, способствовали развитию других языков программирования. Для того чтобы этот механизм работал, компилятор должен поддерживать поиск статичных типов и методов для сопоставления с методами расширения. И компиляторы должны это проделывать быстро, чтобы время компиляции оставалось минимальным. В языке С#, когда вы помечаете первый параметр статичного метода ключевым словом this, компилятор применяет соответствующий атрибут к методу, и данный атрибут сохраняется в метаданных результирующего файла. Этот атрибут определен в сборке System.Core.dll и выглядит следующим образом:



К тому же этот атрибут применяется к метаданным любого статического класса, содержащего, по крайней мере, один метод расширения. Итак, когда скомпилированный код вызывает несуществующий экземплярный метод, компилятор может быстро просканировать все ссылающиеся сборки, чтобы определить, какая из них содержит методы расширения. В дальнейшем он может сканировать только те сборки статических классов, которые содержат методы расширения, выполняя поиск потенциальных соответствий компилируемому коду настолько быстро, насколько это возможно.

## 5.7 Частичные методы

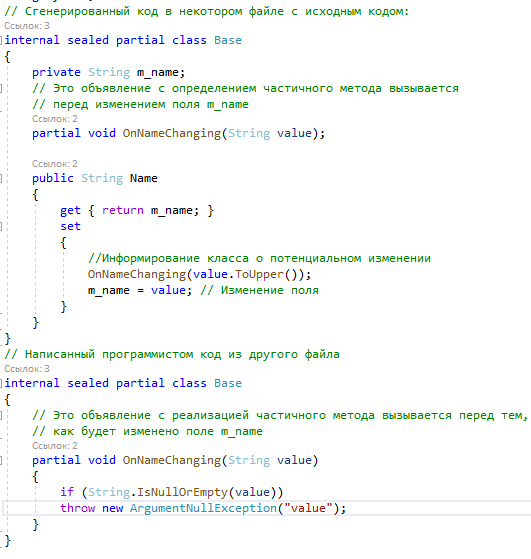
Представьте, что вы используете служебную программу, которая генерирует исходный код на C# с определением типа. Этой программе известно, что внутри программного кода есть места, в которых вы хотели бы настроить поведение типа. Обычно такая настройка производится при помощи виртуальных методов, вызываемых сгенерированным кодом. Сгенерированный код также должен содержать определения этих виртуальных методов, где их реализация ничего не делает, а просто возвращает управление. Чтобы настроить поведение класса, нужно определить собственный класс, унаследованный от базового, и затем переопределить все его виртуальные методы, реализующие желаемое поведение. Вот пример:



К сожалению, у представленного кода имеются два недостатка.

Тип не должен быть запечатанным (sealed) классом. Нельзя использовать этот подход для запечатанных классов или для значимых типов (потому что значимые типы неявно запечатаны). К тому же нельзя использовать этот подход для статических методов, потому что они не могут переопределяться.

Существует проблема эффективности. Тип, определяемый только для переопределения метода, понапрасну расходует некоторое количество системных ресурсов. И даже если вы не хотите переопределять поведение типа OnNameChanging, код базового класса по-прежнему вызовет виртуальный метод, который помимо возврата управления ничего больше не делает. Метод ToUpper вызывается и тогда, когда OnNameChanging получает доступ к переданным аргументам, и тогда, когда не получает. Для решения проблемы переопределения поведения можно задействовать частичные методы языка C#. В следующем коде для достижения той же семантики, что и в предыдущем коде, используются частичные методы:



В этом коде есть несколько мест, на которые необходимо обратить внимание.

Теперь класс запечатан (хотя это и не обязательно). В действительности, класс мог бы быть статическим классом или даже значимым типом.

Код, сгенерированный программой, и код, написанный программистом, на самом деле являются двумя частичными определениями, которые в конце концов образуют одно определение типа (подробности см. в главе 6).

Код, сгенерированный программой, представляет собой объявление частичного метода. Этот метод помечен ключевым словом partial и не имеет тела.

Код, написанный программистом, реализует объявление частичного метода. Этот метод также помечен ключевым словом partial и тоже не имеет тела. Когда вы скомпилируете этот код, вы увидите то же самое, что и в представленном ранее коде. Большое преимущество такого решения заключается в том, что вы можете перезапустить программу и сгенерировать новый код в новом файле, а ваш программный код по-прежнему останется нетронутым в отдельном файле. Кроме того, этот подход работает для изолированных классов, статических классов и значимых типов.

ПримечАние

В редакторе Visual Studio, если ввести partial и нажать пробел, в окне IntelliSense появятся объявления всех частичных методов вложенного типа, которые пока не имеют соответствия объявлениям выполняемого частичного метода. Вы легко можете выбрать частичный метод в IntelliSense-окне, и Visual Studio сгенерирует прототип метода автоматически. Это очень удобная функция, повышающая производительность программирования. У частичных методов имеется еще одно серьезное преимущество. Скажем, у вас теперь нет нужны модифицировать поведение типа, сгенерированного инструментом, и менять файл исходного кода. Если просто скомпилировать такой код, компилятор создаст IL-код и метаданные, как если бы сгенерированный программой код выглядел следующим образом:



При отсутствии объявления выполняемого частичного метода компилятор не будет генерировать метаданные, представляющие частичный метод. К тому же компилятор не сгенерирует IL-команды вызова частичного метода, он не сгенерирует код, вычисляющий аргументы, которые необходимо передать частичному методу. В приведенном примере компилятор не сгенерирует код для вызова метода ToUpper. В результате будет меньше метаданных и IL-кода и производительность во время выполнения повысится!

ПримечАние

Подобным образом частичные методы работают с атрибутом System.Diagnostics. ConditionalAttribute. Однако они работают только с одним типом, тогда как атрибут ConditionalAttribute может быть использован для необязательного вызова методов, определенных в другом типе.

### 5.7.1 Правила и рекомендации

Несколько дополнительных правил и рекомендаций, касающихся частичных методов.

Частичные методы могут объявляться только внутри частичного класса или структуры.

Частичные методы должны всегда иметь возвращаемый тип void и не могут иметь параметров, помеченных ключевым словом out. Эти ограничения связаны с тем, что во время выполнения программы метода не существует, и вы не можете инициализировать переменную, возвращаемую методом, потому что этого метода не существует. По той же причине нельзя использовать параметр, помеченный словом out, потому что иначе метод должен будет инициализировать этот параметр, но этого метода не существует. Частичный метод может иметь параметры, помеченные ключевым словом ref, а также универсальные параметры, экземплярные или статические, или даже параметры, помеченные как unsafe.

Естественно, определяющее объявление частичного метода и его реализующее объявление должны иметь идентичные сигнатуры. И оба должны иметь настраивающиеся атрибуты, применяющиеся к ним, когда компилятор объединяет атрибуты обоих методов вместе. Все атрибуты, применяемые к параметрам, тоже объединяются.

Если не существует реализующего объявления частичного метода, в вашем коде не может быть попыток создания делегата, ссылающегося на частичный метод. Это причина, по которой метод не существует во время выполнения программы. Компилятор выдаст следующее сообщение (ошибка CS0762: не могу создать делегата из метода 'Base.OnNameChanging(string)', потому что это частичный метод без реализующего объявления):

"error CS0762: Cannot create delegate from method 'Base.OnNameChanging(string)' because it is a partial method without an implementing declaration

Хотя частичные методы всегда считаются закрытыми, компилятор C# запрещает писать ключевое слово private перед объявлением частичного метода.

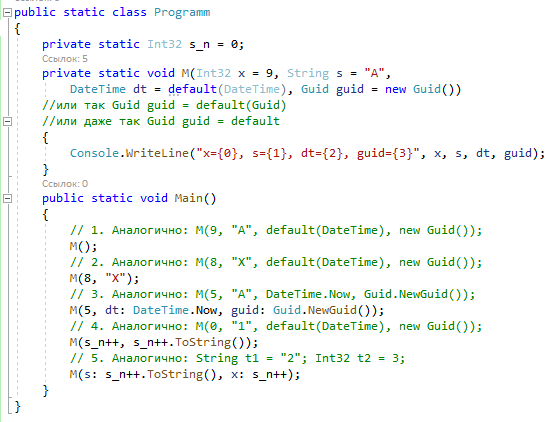
# 6 Параметры

В этой главе рассмотрены различные способы передачи параметров в метод. В числе прочего вы узнаете, как определить необязательный параметр, задать параметр по имени и передать его по ссылке. Также рассмотрена процедура определения методов, принимающих различное количество аргументов.

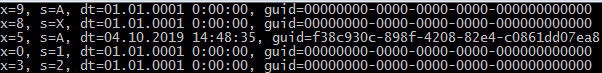
## 6.1 Необязательные и именованные параметры (аргументы)

В настоящий момент в msdn.microsoft.com используется термин «необязательные и именованные аргументы», но автор в данной книге называет их «параметрами». Мы решили сохранить авторскую терминологию.

При выборе параметров метода некоторым из них (и даже всем) можно присваивать значения по умолчанию. В результате в вызывающем такой метод коде можно не указывать эти аргументы, а принимать уже имеющиеся значения. Кроме того, при вызове метода существует возможность указать аргументы, воспользовавшись именами их параметров. Следующий код демонстрирует применение как необязательных, так и именованных параметров:



При выполнении этого кода выводится следующий результат:



Как видите, в случае если при вызове метода аргументы отсутствуют, компилятор берет их значения, предлагаемые по умолчанию. В третьем и пятом вызовах метода M заданы именованные параметры (named parameter). Я в явном виде передал значение переменной x и указал, что хочу передать аргумент для параметров guid и dt. Передаваемые в метод аргументы компилятор рассматривает слева направо. В четвертом вызове метода M значение аргумента s\_n (0) передается в переменную x, затем s\_n увеличивается на единицу и аргумент s\_n (1) передается как строка в параметр s. После чего s\_n увеличивается до 2. Передача аргументов с помощью именованных параметров опять же осуществляется компилятором слева направо. В пятом вызове метода M значение параметра s\_n (2) преобразуется в строку и сохраняется в созданной компилятором временной переменной (t1). Затем s\_n увеличивается до 3, и это значение сохраняется в еще одной созданной компилятором временной переменной (t2). После этого s\_n увеличивается до 4. В конце концов, вызывается метод M, в который передаются переменные t2, t1, переменная DateTime со значением по умолчанию и новое значение Guid

### 6.1.1 Правила использования параметров

Определяя метод, задающий для части своих параметров значения по умолчанию, следует руководствоваться следующими правилами:

Значения по умолчанию указываются для параметров методов, конструкторов методов и параметрических свойств (индексаторов C#). Также их можно указывать для параметров, являющихся частью определения делегатов. В результате при вызове этого типа делегата аргументы можно опускать, используя их значения по умолчанию.

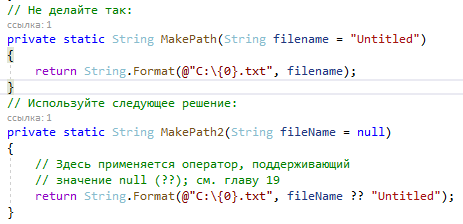
Параметры со значениями по умолчанию должны следовать за всеми остальными параметрами. Другими словами, если указан параметр со значением по умолчанию, значения по умолчанию должны иметь и все параметры, расположенные справа от него. Например, если при определении метода M удалить значение по умолчанию ("A") для параметра s, компилятор выдаст сообщение об ошибке. Существует только одно исключение из правил — параметр массива, помеченный ключевым словом params (о котором мы подробно поговорим чуть позже). Он должен располагаться после всех прочих параметров, в том числе имеющих значение по умолчанию. При этом сам массив значения по умолчанию иметь не может.

Во время компиляции значения по умолчанию должны оставаться неизменными. То есть задавать значения по умолчанию можно для параметров примитивных типов, перечисленных в табл. 5.1 главы 5. Сюда относятся также перечислимые типы и ссылочные типы, допускающие присвоение значения null. Для параметров произвольного значимого типа значение по умолчанию задается как экземпляр этого типа с полями, содержащими нули. Можно использовать как ключевое слово default, так и ключевое слово new, в обоих случаях генери-руется одинаковый IL-код. С примерами обоих вариантов синтаксиса мы уже встречались в методе M при задании значений по умолчанию для параметров dt и guid соответственно.

Запрещается переименовывать параметрические переменные, так как это влечет за собой необходимость редактирования вызывающего кода, который передает аргументы по имени параметра. Скажем, если в объявлении метода M переименовать переменную dt в dateTime, то третий вызов метода станет причиной появления следующего сообщения компилятора

(ошибка CS1739: в подходящей перегруженной версии 'M' отсутствует параметр с именем 'dt'): "error CS1739: The best overload for 'M' does not have a parameter named 'dt'

При вызове метода извне модуля изменение значения параметров по умолчанию является потенциально опасным. Вызывающая сторона использует значение по умолчанию в процессе работы. Если изменить его и не перекомпилировать код, содержащий вызов, в вызываемый метод будет передано прежнее значение. В качестве индикатора поведения можно использовать значение по умолчанию 0 или null. В результате исчезает необходимость повторной компиляции кода вызывающей стороны. Вот пример:



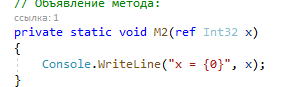
Для параметров, помеченных ключевыми словами ref или out, значения по умолчанию не задаются. Существуют также дополнительные правила вызова методов с использованием необязательных или именованных параметров:

Аргументы можно передавать в произвольном порядке; но именованные аргументы должны находиться в конце списка.

Передача аргумента по имени возможна для параметров, не имеющих значения по умолчанию, но при этом компилятору должны быть переданы все аргументы, необходимые для компиляции (c указанием их позиции или имени).

В C# между запятыми не могут отсутствовать аргументы. Иначе говоря, запись M(1, ,DateTime.Now) недопустима, так как ведет к нечитабельному коду. Чтобы опустить аргумент для параметра со значением по умолчанию, передавайте аргументы по именам параметров.

Вот как передать аргумент по имени параметра, требующего ключевого слова ref/out:



ПримечАние

Синтаксис необязательных и именованных параметров в C# весьма удобен при написании кода, поддерживающего объектную модель COM из Microsoft Office. При вызове COM-компонентов C# позволяет опускать ключевые слова ref/out в процессе передачи аргументов по ссылке. Это еще больше упрощает код. Если же COM-компонент не вызывается, наличие рядом с аргументом ключевого слова ref/out обязательно.

## 6.2 Неявно типизированные локальные переменные

3. Операторы языка C#

3.1. Арифметические

3.2. Отношения

3.3. Логические

3.4. Присваивания

3.5. Поразрядные

3.6. Оператор

3.7. Использование скобок

3.8. Деление

4. Операторы языка C#

4.1. Управляющие операторы

4.2. Операторы переходов

4.3. Операторы проверки условий

4.4. Операторы циклов

5. Массивы и строки

5.1. Начальные сведения о массивах

5.2. Одномерные массивы

5.3. Многомерные массивы

5.4. Ступенчатые массивы

5.5. Строки

6. Коллекции

6.1. Основы работы со стандартными коллекциями

6.2. Коллекции-списки

6.3. Коллекции-словари

6.4. Пользовательские коллекции

7. Введение в классы

7.1. Синтаксис объявления класса

7.3. Использование параметров

7.5. Разделяемые классы и методы

8. Введение в классы

8.1. Свойства и индексаторы

8.4. Методы расширения

8.6. Рекурсия

8.7. Стратегия поиска ошибок и отладка кода

8.8. Области видимости переменных

## 8.9. Класс System.Object

Все типы должны быть производными (прямо или опосредованно) от предопределенного типа System.Object (то есть от типа Object из пространства имен System). Тип Object является корнем иерархии типов, а следовательно, гарантирует, что каждый экземпляр типа обладает минимальным набором аспектов поведения. А если говорить конкретнее, тип System.Object позволяет сделать следующее:

сравнить два экземпляра на равенство Equals;

получить хеш-код экземпляра GetHashCode;

запросить фактический тип экземпляра GetType;

выполнить поверхностное (поразрядное) копирование экземпляра MemberwiseClone;

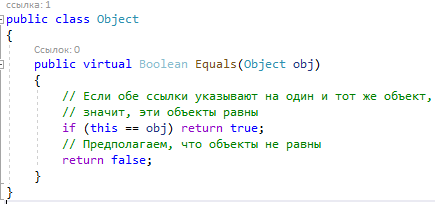
получить строковое представление текущего состояния экземпляра ToString

Класс System.Object определен в MSCorLib.dll.

### 8.9.1 Равенство и тождество объектов Equals

Часто разработчикам приходится создавать код сравнения объектов. В частности, это необходимо, когда объекты размещаются в коллекциях и требуется писать код для сортировки, поиска и сравнения отдельных элементов в коллекции.

У типа System.Object есть виртуальный метод Equals, который возвращает true для двух «равных» объектов. Вот как выглядит реализация метода Equals для Object:



На первый взгляд эта реализация выглядит вполне разумно: сравниваются две ссылки, переданные в аргументах this и obj, и если они указывают на один объект, возвращается true, в противном случае возвращается false. Это кажется логичным, так как Equals «понимает», что объект равен самому себе. Однако если аргументы ссылаются на разные объекты, Equals сложнее определить, содержат ли объекты одинаковые значения, поэтому возвращается false. Иначе говоря, оказывается, что стандартная реализация метода Equals типа Object реализует проверку на тождество, а не на равенство значений.

Вот как должна действовать правильная реализация метода Equals:

1. Если аргумент obj равен null, вернуть false, так как ясно, что текущий объект, указанный в this, не равен null при вызове нестатического метода Equals.

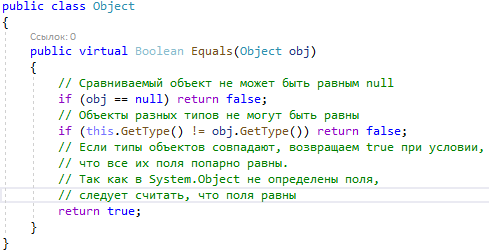
2. Если аргументы obj и this ссылаются на объекты одного типа, вернуть true. Этот шаг поможет повысить производительность в случае сравнения объектов с многочисленными полями.

3. Если аргументы obj и this ссылаются на объекты разного типа, вернуть false. Понятно, что результат сравнения объектов String и FileStream равен false.

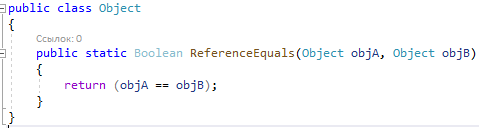
4. Сравнить все определенные в типе экземплярные поля объектов obj и this. Если хотя бы одна пара полей не равна, вернуть false.

5. Вызвать метод Equals базового класса, чтобы сравнить определенные в нем поля. Если метод Equals базового класса вернул false, тоже вернуть false, в противном случае вернуть true.

Учитывая это, компания Microsoft должна была бы реализовать метод Equals типа Object примерно так:



Однако, поскольку в Microsoft метод Equals реализован иначе, правила собственной реализации Equals намного сложнее, чем кажется. Если ваш тип переопределяет Equals, переопределенная версия метода должна вызывать реализацию Equals базового класса, если только не планируется вызывать реализацию в типе Object. Это означает еще и то, что поскольку тип может переопределять метод Equals типа Object, этот метод больше не может использоваться для проверки на тождественность. Для исправления ситуации в Object предусмотрен статический метод ReferenceEquals со следующим прототипом:



Для проверки на тождественность нужно всегда вызывать ReferenceEquals (то есть проверять на предмет того, относятся ли две ссылки к одному объекту). Не нужно использовать оператор == языка C# (если только перед этим оба операнда не приводятся к типу Object), так как тип одного из операндов может перегружать этот оператор, в результате чего его семантика перестает соответствовать понятию «тождественность». Как видите, в области равенства и тождественности в .NET Framework дела обстоят довольно сложно. Кстати, в System.ValueType (базовом классе всех значимых типов) метод Equals типа Object переопределен и корректно реализован для проверки на равенство (но не тождественность). Внутреняя реализация переопределенного метода работает по следующей схеме:

1. Если аргумент obj равен null, вернуть false.

2. Если аргументы obj и this ссылаются на объекты разного типа, вернуть false.

3. Для каждого экземплярного поля, определенного типом, сравнить значение из объекта obj со значением из объекта this вызовом метода Equals поля. Если хотя бы одна пара полей не равна, вернуть false.

4. Вернуть true. Метод Equals типа ValueType не вызывает одноименный метод типа Object. Для выполнения шага 3 в методе Equals типа ValueType используется отражение (см. главу 23). Так как отражение в CLR работает медленно, при создании собственного значимого типа нужно переопределить Equals и создать свою реализацию, чтобы повысить производительность сравнения значений на предмет равенства экземпляров созданного типа. И, конечно же, не стоит вызывать из этой реализации метод Equals базового класса. Определяя собственный тип и приняв решение переопределить Equals, обеспечьте поддержку четырех характеристик, присущих равенству:

Рефлексивность: x.Equals(x) должно возвращать true.

Симметричность: x.Equals(y) и y.Equals(x) должны возвращать одно и то же значение.

Транзитивность: если x.Equals(y) возвращает true и y.Equals(z) возвращает true, то x.Equals(z) также должно возвращать true.

Постоянство: если в двух сравниваемых значениях не произошло изменений, результат сравнения тоже не должен измениться. Отступление от этих правил при создании собственной реализации Equals грозит непредсказуемым поведением приложения. При переопределении метода Equals может потребоваться выполнить несколько дополнительных операций.

Реализовать в типе метод Equals интерфейса System.IEquatable. Этот обобщенный интерфейс позволяет определить безопасный в отношении типов метод Equals. Обычно Equals реализуют так, что, принимая параметр типа Object, код метода вызывает безопасный в отношении типов метод Equals.

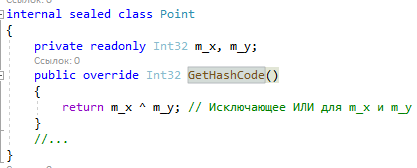
Перегрузить методы операторов == и !=. Обычно код реализации этих операторных методов вызывает безопасный в отношении типов метод Equals. Если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется также реализовать метод CompareTo типа System.Icomparable и безопасный в отношении типов метод CompareTo типа System.IComparable. Реализовав эти методы, можно реализовать метод Equals так, чтобы он вызывал CompareTo типа System.IComparable и возвращал true, если CompareTo возвратит 0. После реализации методов CompareTo также часто требуется перегрузить методы различных операторов сравнения (<, <=, >, >=) и реализовать код этих методов так, чтобы он вызывал безопасный в отношении типов метод CompareTo.

### 8.9.2 Хеш-коды объектов GetHashCode

Виртуальный метод GetHashCode, позволяющий вычислить для любого объекта целочисленный (Int32) хеш-код. Если вы определяете тип и переопределяете метод Equals, вы должны переопределить и метод GetHashCode. Если при определении типа переопределить только один из этих методов, компилятор C# выдаст предупреждение. Например, при компиляции представленного далее кода появится предупреждение:

warning CS0659: 'Program' overrides Object.Equals(Object o) but does not override Object.GetHashCode() ('Program' переопределяет Object.Equals(Object o), но не переопределяет Object.GetHashCode()).

Причина, по которой в типе должны быть определены оба метода — Equals и GetHashCode, — состоит в том, что реализация типов System.Collections.Hashtable, System.Collections.Generic.Dictionary и любых других коллекций требует, чтобы два равных объекта имели одинаковые значения хеш-кодов. Поэтому, переопределяя Equals, нужно переопределить GetHashCode и обеспечить соответствие алгоритма, применяемого для вычисления равенства, алгоритму, используемому для вычисления хеш-кода объекта. По сути, когда вы добавляете пару «ключ-значение» в коллекцию, первым вычисляется хеш-код ключа. Он указывает, в каком «сегменте» будет храниться пара «ключ-значение». Когда коллекции требуется найти некий ключ, она вычисляет для него хеш-код. Хеш-код определяет «сегмент» поиска имеющегося в таблице ключа, равного заданному. Применение этого алгоритма хранения и поиска ключей означает, что если вы измените хранящийся в коллекции ключ объекта, коллекция больше не сможет найти этот объект. Если вы намерены изменить ключ объекта в хеш-таблице, то сначала удалите имеющуюся пару «ключ-значение», модифицируйте ключ, а затем добавьте в хеш-таблицу новую пару «ключ-значение». В определении метода GetHashCode нет особых хитростей. Однако для некоторых типов данных и их распределения в памяти бывает непросто подобрать алгоритм хеширования, который выдавал бы хорошо распределенный диапазон значений. Вот простой алгоритм, неплохо подходящий для объектов Point:



Выбирая алгоритм вычисления хеш-кодов для экземпляров своего типа, старайтесь следовать определенным правилам:

Используйте алгоритм, который дает случайное распределение, повышающее производительность хеш-таблицы.

Алгоритм может вызывать метод GetHashCode базового типа и использовать возвращаемое им значение, однако в общем случае лучше отказаться от вызова встроенного метода GetHashCode для типа Object или ValueType, так как эти реализации обладают низкой производительностью алгоритмов хеширования.

В алгоритме должно использоваться как минимум одно экземплярное поле.

Поля, используемые в алгоритме, в идеале не должны изменяться, то есть они должны инициализироваться при создании объекта и сохранять значение в течение всей его жизни.

Алгоритм должен быть максимально быстрым. Объекты с одинаковым значением должны возвращать одинаковые коды. Например, два объекта String, содержащие одинаковый текст, должны возвращать одно значение хеш-кода. Реализация GetHashCode в System.Object ничего «не знает» о производных типах и их полях. Поэтому этот метод возвращает число, однозначно идентифицирующее объект в пределах домена приложений; при этом гарантируется, что это число не изменится на протяжении всей жизни объекта.

8.10. Жизненный цикл объекта

8.11. Структуры

8.12. Перечисления

# Глава 3 Курс программирования на C# Основы ООП (C Sharp)

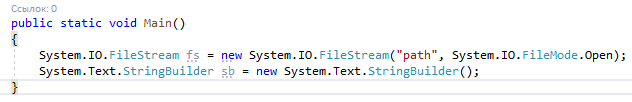
# 1. Основы ООП

1.1. Принципы ООП

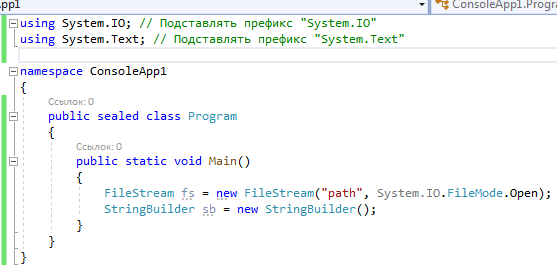
1.2. Инкапсуляция

## 1.3. Пространство имен

Пространства имен используются для логической группировки родственных типов, чтобы разработчику было проще найти нужный тип. Например, в пространстве имен System.Text описаны типы для обработки строк, а в пространстве имен System.IO — типы для выполнения операций ввода-вывода.

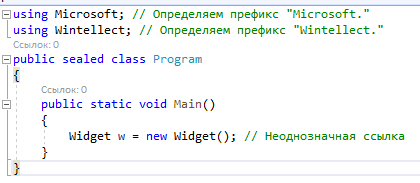


К счастью, многие компиляторы предоставляют программистам механизмы, позволяющие сократить объем набираемого текста. Например, в компиляторе C# предусмотрена директива using. Следующий код аналогичен предыдущему:

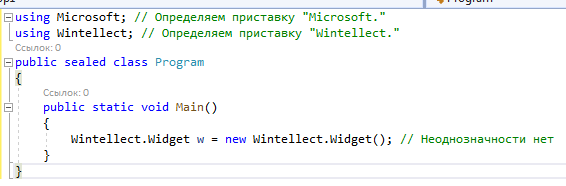


Для компилятора пространство имен — простое средство, позволяющее удлинить имя типа и сделать его уникальным за счет добавления к началу имени групп символов, разделенных точками. Например, в данном примере компилятор интерпретирует FileStream как System.IO.FileStream, а StringBuilder — как System. Text.StringBuilder. Применять директиву using в C# не обязательно, при необходимости достаточно ввести полное имя типа. Директива using заставляет компилятор C# добавлять к имени указанный префикс, пока не будет найдено совпадение.

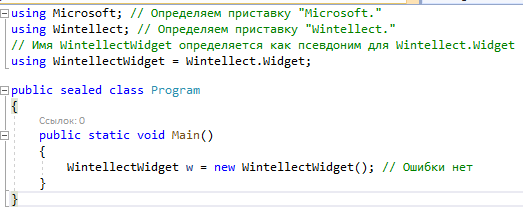
Легко догадаться, что такой способ обработки пространства имен чреват проблемами, если два (и более) типа с одинаковыми именами находятся в разных сборках. Microsoft настоятельно рекомендует при описании типов применять уникальные имена. Но порой это невозможно. В следующем коде ссылка на Widget неоднозначна, и компилятор C# выдаст сообщение error CS0104: 'Widget' is an ambiguous reference (ошибка CS0104: 'Widget' — неоднозначная ссылка):



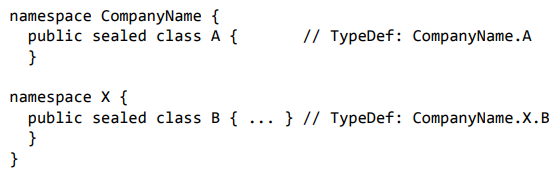
Для того чтобы избавиться от неоднозначности, надо явно указать компилятору, какой экземпляр Widget требуется создать:

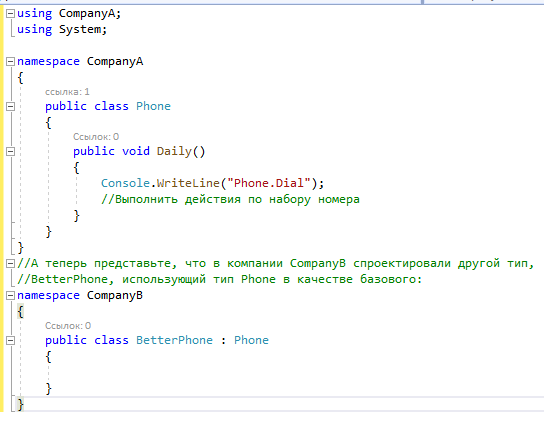


В C# есть еще одна форма директивы using, позволяющая создать псевдоним для отдельного типа или пространства имен. Она удобна, если вы намерены использовать несколько типов из пространства имен, но не хочется «захламлять» глобальное пространство имен всеми используемыми типами. Альтернативный способ преодоления неоднозначности следующий:



Эти методы устранения неоднозначности хороши, но иногда их недостаточно. Представьте, что компании Australian Boomerang Company (ABC) и Alaskan Boat Corporation (ABC) создали каждая свой тип с именем BuyProduct и собираются поместить его в соответствующие сборки. Обе компании создадут пространства имен ABC, в которые и включат тип BuyProduct. К счастью, в компиляторе C# поддерживаются внешние псевдонимы (extern aliases), позволяющие справиться с проблемой. Внешние псевдонимы дают также возможность обращаться к одному типу двух (или более) версий одной сборки. При проектировании типов, применяемых в библиотеках, которые могут использоваться третьими лицами, старайтесь описывать эти типы в пространстве имен так, чтобы компиляторы могли без труда преодолеть неоднозначность типов. Вероятность конфликта заметно снизится, если в пространстве имен верхнего уровня указывается полное, а не сокращенное название компании. В документации .NET Framework SDK Microsoft использует для своих типов пространство имен Microsoft (например: Microsoft.CSharp, Microsoft.VisualBasic и Microsoft.Win32). Создавая пространство имен, включите в код его объявление (на C#):





1.4. Перегрузка операций

2. Наследование классов

2.1. Основы наследования

2.2. Конструкторы и наследование

2.3. Виртуальные методы

2.4. Полиморфизм

2.5. Абстрактные классы

3. Классы, интерфейсы

3.1. Статические классы

3.2. Статик конструкторы

3.3. Абстрактные классы

3.4. Абстрактные компоненты

3.5. Интерфейсы

3.6. Наследование интерфейсов

4. Сборка мусора

4.1. Оператор new

4.2. Работа сборщика мусора

4.3. Поколения

4.4. Применение деструкторов

5. Делегаты и события

5.1. Делегаты

5.2. Анонимные функции

5.3. Лямбда-выражения

5.4. События

6. Генерация и обработка исключительных ситуаций

6.1. Основы обработки исключительных ситуаций

6.2. Последствия не перехвата исключений

6.3. Перехват исключений

7. Работа с объектами файловой системы

7.1. Использование потоков данных

7.2. Классы для работы с потоками

7.3. Основы XML

7.4. Сериализация

8. Основы многопоточного программирования

8.1. Многопоточность

8.2. Синхронизация потоков (lock, mutex, семафор)

8.3. Процессы и домены

8.4. TPL

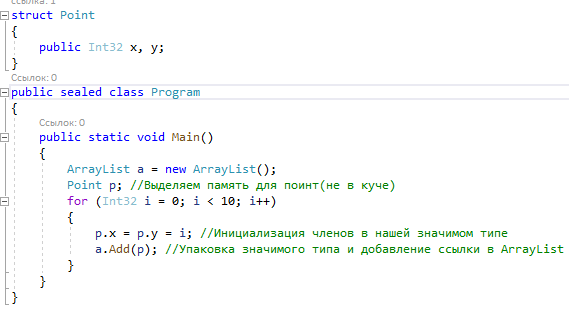
9. Перечислители и итераторы, визуальная разработка

9.1. Универсальные шаблоны

9.2. Класс System.Object

## 9.3. Операции упаковки, распаковки

### 9.3.1 Упаковка значимых типов

Значимые типы «легче» ссылочных: для них не нужно выделять память в управляемой куче, их не затрагивает сборка мусора, к ним нельзя обратиться через указатель. Однако часто требуется получать ссылку на экземпляр значимого типа, например если вы хотите сохранить структуры Point в объекте типа ArrayList (определен в пространстве имен System.Collections). В коде это выглядит примерно следующим образом:

В каждой итерации цикла инициализируются поля значимого типа Point, после чего Point помещается в ArrayList. Задумаемся, что же помещается в ArrayList: сама структура Point, адрес структуры Point или что-то иное? За ответом обратимся к методу Add типа ArrayList и посмотрим описание его параметра. В данном случае прототип метода Add выглядит следующим образом:

public virtual Int32 Add(Object value);

Отсюда видно, что в параметре Add должен передаваться тип Object, то есть ссылка (или указатель) на объект в управляемой куче. Однако в примере я передаю переменную p, имеющую значимый тип Point. Чтобы код работал, нужно преобразовать значимый тип Point в объект из управляемой кучи и получить на него ссылку. Для преобразования значимого типа в ссылочный служит упаковка (boxing). При упаковке экземпляра значимого типа происходит следующее.

1. В управляемой куче выделяется память. Ее объем определяется длиной значимого типа и двумя дополнительными членами — указателем на типовой объект и индексом блока синхронизации. Эти члены необходимы для всех объектов в управляемой куче.

2. Поля значимого типа копируются в память, только что выделенную в куче.

3. Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть значимый тип превращается в ссылочный. Компилятор C# создает IL-код, необходимый для упаковки экземпляра значимого типа, автоматически, но вы должны понимать, что происходит «за кулисами» и помнить об опасности «распухания» кода и снижения производительности. В предыдущем примере компилятор C# обнаружил, что методу, требующему ссылочный тип, в параметре передается значимый тип, и автоматически создал код для упаковки объекта. Вследствие этого поля экземпляра p значимого типа Point в период выполнения копируются во вновь созданный в куче объект Point. Полученный адрес упакованного объекта Point (теперь это ссылочный тип) передается методу Add. Объект Point остается в куче до очередной уборки мусора. Переменную p значимого типа Point можно использовать повторно, так как ArrayList ничего о ней не знает. Заметьте: время жизни упакованного значимого типа превышает время жизни неупакованного значимого типа.

### 9.3.2 Распаковка значимых типов

Познакомившись с упаковкой, перейдем к распаковке. Допустим, в другом месте кода нужно извлечь первый элемент массива ArrayList: 

Здесь ссылка (или указатель), содержащаяся в элементе с номером 0 массива ArrayList, помещается в переменную p1 значимого типа Point. Для этого все поля, содержащиеся в упакованном объекте Point, надо скопировать в переменную p1 значимого типа, находящуюся в стеке потока. CLR выполняет эту процедуру в два этапа. Сначала извлекается адрес полей Point из упакованного объекта Point. Этот процесс называют распаковкой (unboxing). Затем значения полей копируются из кучи в экземпляр значимого типа, находящийся в стеке. Распаковка не является точной противоположностью упаковки. Она гораздо менее ресурсозатратна, чем упаковка, и состоит только в получении указателя на исходный значимый тип (поля данных), содержащийся в объекте. В сущности, указатель ссылается на неупакованную часть упакованного экземпляра, и никакого копирования при распаковке (в отличие от упаковки) не требуется. Однако вслед за распаковкой обычно выполняется копирование полей. Понятно, что упаковка и распаковка/копирование снижают производительность приложения (в плане как замедления, так и расходования дополнительной памяти), поэтому нужно знать, когда компилятор сам создает код для выполнения этих операций, и стараться свести их к минимуму. При распаковке упакованного значимого типа происходит следующее.

1. Если переменная, содержащая ссылку на упакованный значимый тип, равна null, генерируется исключение NullReferenceException.

2. Если ссылка указывает на объект, не являющийся упакованным значением требуемого значимого типа, генерируется исключение InvalidCastException

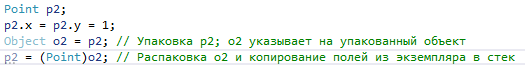
Из второго пункта следует, что приведенный ниже код не работает так, как хотелось бы:



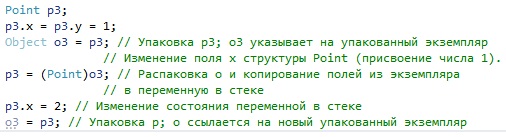
Казалось бы, можно взять упакованный экземпляр Int32, на который указывает o, и привести к типу Int16. Однако при распаковке объекта должно быть выполнено приведение к неупакованному типу (в нашем случае — к Int32). Вот как выглядит правильный вариант:



Как я уже отмечал, распаковка часто сопровождается копированием полей. Следующий код на C# демонстрирует, что операции распаковки и копирования часто работают совместно:

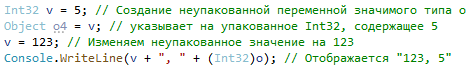


В последней строке компилятор C# генерирует IL-команду для распаковки o (получение адреса полей в упакованном экземпляре) и еще одну IL-команду для копирования полей из кучи в переменную p, располагающуюся в стеке. Теперь посмотрите на следующий пример:



Во второй части примера нужно изменить поле x структуры Point с 1 на 2. Для этого выполняют распаковку, копирование полей, изменение поля (в стеке) и упаковку (создающую новый объект в управляемой куче). Вероятно, вы понимаете, что все эти операции обязательно сказываются на производительности приложения.

Рассмотрим еще несколько примеров, демонстрирующих упаковку и распаковку:



Сколько в этом коде операций упаковки и распаковки? Вы не поверите — целых три!

Вначале в стеке создается экземпляр v неупакованного значимого типа Int32, которому присваивается число 5. Затем создается переменная o4 типа Object, которая инициализируется указателем на v. Однако поскольку ссылочные типы всегда должны указывать на объекты в куче, C# генерирует соответствующий IL-код для упаковки v и заносит адрес упакованной «копии» v в o4. Теперь величина 123 помещается в неупакованный значимый тип v, но это не влияет на упакованное значение типа Int32, которое остается равным 5. Дальше вызывается метод WriteLine, которому нужно передать объект String, но такого объекта нет. Вместо строкового объекта мы имеем неупакованный экземпляр значимого типа Int32 (v), объект String (ссылочного типа) и ссылку на упакованный экземпляр значимого типа Int32 (o), который приводится к неупакованному типу Int32. Эти элементы нужно как-то объединить, чтобы получился объект String. Чтобы создать String, компилятор C# формирует код, в котором вызывается статический метод Concat объекта String. Есть несколько перегруженных версий этого метода, различающихся лишь количеством параметров. Поскольку строка формируется путем конкатенации трех элементов, компилятор выбирает следующую версию метода Concat:

public static String Concat(Object arg0, Object arg1, Object arg2);

В качестве первого параметра, arg0, передается v. Но v — это неупакованное значение, а arg0 — это значение Object, поэтому экземпляр v нужно упаковать, а его адрес передать в качестве arg0. Параметром arg1 является строка "," в виде ссылки на объект String. И наконец, чтобы передать параметр arg2, o (ссылка на Object) приводится к типу Int32. Для этого нужна распаковка (но без копирования), при которой извлекается адрес неупакованного экземпляра Int32 внутри упакованного экземпляра Int32. Этот неупакованный экземпляр Int32 надо опять упаковать, а его адрес в памяти передать в качестве параметра arg2 методу Concat. Метод Concat вызывает методы ToString для каждого указанного объекта и выполняет конкатенацию строковых представлений этих объектов. Возвращаемый из Concat объект String передается затем методу WriteLine, который отображает окончательный результат. Полученный IL-код станет эффективнее, если обращение к WriteLine переписать:

Console.WriteLine(v + ", " + o); // Отображается "123, 5"

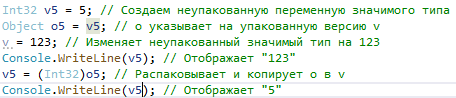
Этот вариант строки отличается от предыдущего только отсутствием для переменной o операции приведения типа (Int32). Этот код выполняется быстрее, так как o уже является ссылочным типом Object и его адрес можно сразу передать методу Concat. Отказавшись от приведения типа, я избавился от двух операций: распаковки и упаковки. В этом легко убедиться, если заново собрать приложение и посмотреть на сгенерированный IL-код:

Беглое сравнение двух версий IL-кода метода Main показывает, что вариант без приведения типа Int32 на 10 байт меньше, чем вариант с приведением типа. Дополнительные операции распаковки/упаковки, безусловно, приводят к разрастанию кода. Если мы пойдем дальше, то увидим, что эти операции потребуют выделения памяти в управляемой куче для дополнительного объекта, которую в будущем должен освободить уборщик мусора. Конечно, обе версии приводят к одному результату и разница в скорости незаметна, однако лишние операции упаковки, выполняемые многократно (например, в цикле), могут заметно повлиять на производительность приложения и расходование памяти.

Предыдущий код можно улучшить, изменив вызов метода WriteLine:



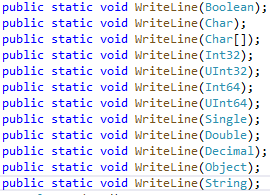
Для неупакованного значимого типа v теперь вызывается метод ToString, возвращающий String. Строковые объекты являются ссылочными типами и могут легко передаваться в метод Concat без упаковки. Вот еще один пример, демонстрирующий упаковку и распаковку:



Сколько операций упаковки вы насчитали в этом коде? Правильно — одну. Дело в том, что в классе System.Console описан метод WriteLine, принимающий в качестве параметра тип Int32:

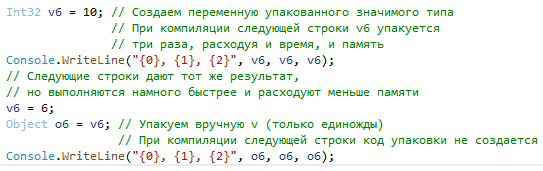
public static void WriteLine(Int32 value);

В показанных ранее вызовах WriteLine переменная v, имеющая неупакованный значимый тип Int32, передается по значению. Возможно, где-то у себя WriteLine упакует это значение Int32, но тут уж ничего не поделаешь. Главное — мы сделали то, что от нас зависело: убрали упаковку из своего кода. Пристально взглянув на FCL, можно заметить, что многие перегруженные методы используют в качестве параметров значимые типы. Так, тип System.Console предлагает несколько перегруженных вариантов метода WriteLine:



Аналогичный набор перегруженных версий есть у метода Write типа System. Console, у метода Write типа System.IO.BinaryWriter, у методов Write и WriteLine типа System.IO.TextWriter, у метода AddValue типа System.Runtime.Serialization.SerializationInfo, у методов Append и Insert типа System.Text. StringBuilder и т. д. Большинство этих методов имеет перегруженные версии только затем, чтобы уменьшить количество операций упаковки для наиболее часто используемых значимых типов.

Если вы определите собственный значимый тип, у этих FCL-классов не будет соответствующей перегруженной версии для вашего типа. Более того, для ряда значимых типов, уже существующих в FCL, нет перегруженных версий указанных методов. Если вызывать метод, у которого нет перегруженной версии для передаваемого значимого типа, результат в конечном итоге будет один — вызов перегруженного метода, принимающего Object. Передача значимого типа как Object приведет к упаковке, что отрицательно скажется на производительности. Определяя собственный класс, можно задать в нем обобщенные методы (возможно, содержащие параметры типа, которые являются значимыми типами). Обобщения позволяют определить метод, принимающий любой значимый тип, не требуя при этом упаковки (см. главу 12). И последнее, что касается упаковки: если вы знаете, что ваш код будет периодически заставлять компилятор упаковывать какой-то значимый тип, можно уменьшить объем и повысить быстродействие своего кода, выполнив упаковку этого типа вручную. Взгляните на следующий пример.



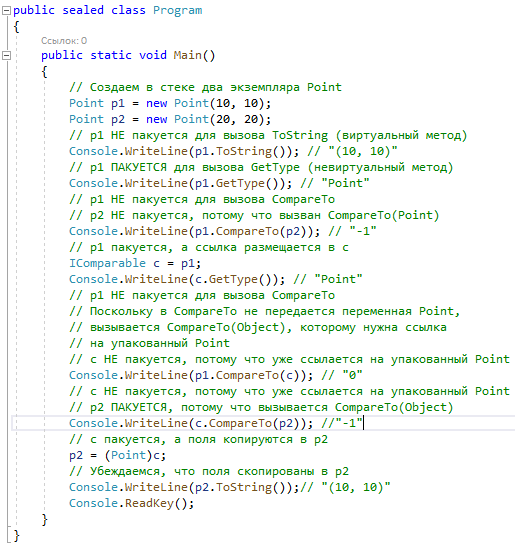
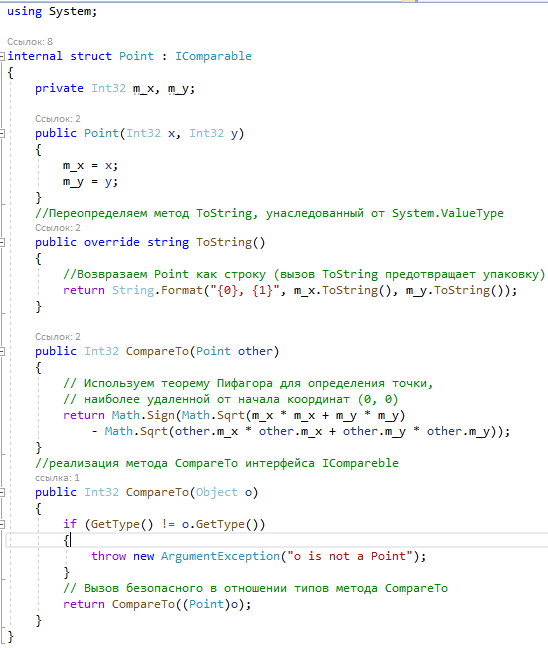
В этих примерах довольно легко определить, где нужно упаковать экземпляр значимого типа. Простое правило:

если нужна ссылка на экземпляр значимого типа, этот экземпляр должен быть упакован. Обычно упаковка выполняется, когда надо передать значимый тип методу, требующему ссылочный тип. Однако могут быть и другие ситуации, когда требуется упаковать экземпляр значимого типа. Помните, мы говорили, что неупакованные значимые типы «легче» ссылочных, поскольку:

память в управляемой куче им не выделяется;

у них нет дополнительных членов, присущих каждому объекту в куче: указателя на типовой объект и индекса блока синхронизации. Поскольку неупакованные значимые типы не имеют индекса блока синхронизации, то не может быть и нескольких потоков, синхронизирующих свой доступ к экземпляру через методы типа System.Threading.Monitor (или инструкция lock языка C#).

Хотя неупакованные значимые типы не имеют указателя на типовой объект, вы все равно можете вызывать виртуальные методы (такие, как Equals, GetHashCode или ToString), унаследованные или пeреопределенные этим типом. Если ваш значимый тип переопределяет один из этих виртуальных методов, CLR может вызвать метод невиртуально, потому что значимые типы неявно запечатываются и поэтому не могут выступать базовыми классами других типов. Кроме того, экземпляр значимого типа, используемый для вызова виртуального метода, не упаковывается. Но если ваше переопределение виртуального метода вызывает реализацию этого метода из базового типа, экземпляр значимого типа упаковывается при вызове реализации базового типа, чтобы в указателе this базового метода передавалась ссылка на объект в куче. Вместе с тем вызов невиртуального унаследованного метода (такого, как GetType или MemberwiseClone) всегда требует упаковки значимого типа, так как эти методы определены в System.Object, поэтому методы ожидают, что в аргументе this передается указатель на объект в куче. Кроме того, приведение неупакованного экземпляра значимого типа к одному из интерфейсов этого типа требует, чтобы экземпляр был упакован, так как интерфейсные переменные всегда должны содержать ссылку на объект в куче. (Об интерфейсах см. главу 13.) Сказанное иллюстрирует следующий код:



В этом примере демонстрируется сразу несколько сценариев поведения кода, связанного с упаковкой/распаковкой.

Вызов ToString. При вызове ToString упаковка p1 не требуется. Казалось бы, тип p1 должен быть упакован, так как ToString — метод, унаследованный от базового типа, System.ValueType. Обычно для вызова виртуального метода нужен указатель на типовой объект, а поскольку p1 является неупакованным значимым типом, то нет ссылки на типовой объект Point. Однако JIT-компилятор видит, что метод ToString переопределен в Point, и создает код, который напрямую (невиртуально) вызывает ToString. Компилятор знает, что полиморфизм здесь невозможен, коль скоро Point является значимым типом, а значимые типы не могут применяться для другого типа в качестве базового и по-другому реализовывать виртуальный метод. Ели бы метод ToString из Point во внутренней реализации вызывал base.ToString(), то экземпляр значимого типа был бы упакован при вызове метода ToString типа System.ValueType.

Вызов GetType. При вызове невиртуального метода GetType упаковка p1 необходима, поскольку тип Point не реализует GetType, а наследует его от System.Object. Поэтому для вызова GetType нужен указатель на типовой объект Point, который можно получить только путем упаковки p1.

Первый вызов CompareTo. При первом вызове CompareTo упаковка p1 не нужна, так как Point реализует метод CompareTo, и компилятор может просто вызвать его напрямую. Заметьте: в CompareTo передается переменная p2 типа Point, поэтому компилятор вызывает перегруженную версию CompareTo, которая принимает параметр типа Point. Это означает, что p2 передается в CompareTo по значению, и никакой упаковки не требуется.

Приведение типа к IComparable. Когда выполняется приведение типа p1 к переменной интерфейсного типа (с), упаковка p1 необходима, так как интерфейсы по определению имеют ссылочный тип. Поэтому выполняется упаковка p1, а указатель на этот упакованный объект сохраняется в переменной c. Следующий вызов GetType подтверждает, что c действительно ссылается на упакованный объект Point в куче.

Второй вызов CompareTo. При втором вызове CompareTo упаковка p1 не производится, потому что Point реализует метод CompareTo, и компилятор может вызывать его напрямую. Заметьте, что в CompareTo передается переменная с интерфейса IComparable, поэтому компилятор вызывает перегруженную версию CompareTo, которая принимает параметр типа Object. Это означает, что передаваемый параметр должен являться указателем, ссылающимся на объект в куче. К счастью, с уже ссылается на упакованный объект Point, по этой причине адрес памяти из c может передаваться в CompareTo и никакой дополнительной упаковки не требуется.

Третий вызов CompareTo. При третьем вызове CompareTo переменная c уже ссылается на упакованный объект Point в куче. Поскольку переменная c сама по себе имеет интерфейсный тип IComparable, можно вызывать только метод CompareTo интерфейса, а ему требуется параметр Object. Это означает, что передаваемый аргумент должен быть указателем, ссылающимся на объект в куче. Поэтому выполняется упаковка p2 и указатель на этот упакованный объект передается в CompareTo

Приведение типа к Point. Когда выполняется приведение c к типу Point, объект в куче, на который указывает c, распаковывается, и его поля копируются из кучи в p2, экземпляр типа Point, находящийся в стеке. Понимаю, что вся эта информация о ссылочных и значимых типах, упаковке и распаковке поначалу выглядит устрашающе. И все же любой разработчик, стремящийся к долгосрочному успеху на ниве .NET Framework, должен хорошо усвоить эти понятия — только так можно научиться быстро и легко создавать эффективные приложения.

9.4. Основы Unit Testing

9.5. Обзор WPF

# Глава 4 Промышленное программирование на ASP.NET (курс EPAM)

# 1. Инструментарий разработчика (нет ответа)

## 1.1 Visual Studio 2017 (нет ответа)

• Инструменты веб разработчика  
• Средства отладки приложений.  
• Использование NuGet пакетов.

* + Системы контроля версий и работа с ними.

• Основы Git.   
• Обзор приложений для работы с git.  
• GitFlow.

* + Тестирование кода и средства для тестирования.

• Модульное тестирование.   
• Интеграционное тестирование.   
• Нагрузочное тестирование.  
• Библиотека Moq.

# 2. Введение в Asp.NET MVC 5

## 1.1 Коротко о браузере

Браузер — это движок рендеринга. Его работа заключается в том, чтобы загрузить веб-страницу и представить её в понятном для человека виде.

**Что делает браузер?**

Разрешение DNS - Этот процесс помогает браузеру узнать, к какому серверу он должен подключиться, когда пользователь вводит URL. Браузер связывается с DNS-сервером и обнаруживает, что google.com соответствует IP-адресу 216.58.207.110, к которому может подключиться браузер;

HTTP-обмен - Как только браузер определит, какой сервер будет обслуживать наш запрос, он установит с ним TCP-соединение и начнет HTTP-обмен. Это не что иное, как способ общения браузера с нужным ему сервером, а для сервера — способ отвечать на запросы браузера. HTTP-обмен подразумевает, что клиент (наш браузер) отправляет запрос, а сервер присылает ответ.

Рендеринг - Последним по счёту, идет процесс рендеринга. В теле ответа сервер включает представление запрашиваемого документа в соответствии с заголовком Content-Type. В нашем случае тип содержимого был установлен на text/html, поэтому мы ожидаем HTML-разметку в ответе — и именно ее мы и находим в теле документа.

Браузер считывает и анализирует HTML-код, загружает дополнительные ресурсы, включенные в разметку (например, там могут быть указаны для подгрузки JavaScript-файлы или CSS-документы) и представляет их пользователю.

Еще раз, конечным результатом должно стать то, что доступно для восприятия человека;

Сброс и повтор.

## 1.2 Знакомство с работой HTTP протокола

HTTP (HyperText Transfer Protocol) – это протокол седьмого уровня [модели OSI](https://zametkinapolyah.ru/veb-programmirovanie/servernoe-programmirovanie-server-apache/osnovy-interneta-chast-1-sem-urovnej-modeli-osi-princip-raboty-etalonnoj-modeli.html) для передачи данных, в основе которого лежит архитектура взаимодействие [клиент-сервер](https://zametkinapolyah.ru/servera-i-protokoly/o-modeli-vzaimodejstviya-klient-server-prostymi-slovami-arxitektura-klient-server-s-primerami.html). Изначально протокол HTTP разрабатывался для передачи [HTML документов](https://zametkinapolyah.ru/verstka-sajtov/struktura-html-dokumenta-tip-html-dokumenta-zagolovok-html-dokumenta-telo-html-dokumenta-granicy-html-dokumenta.html) между [сервером](https://zametkinapolyah.ru/servera-i-protokoly/chto-takoe-server-servernyj-kompyuter-i-servernoe-prilozhenie.html) и [клиентом](https://zametkinapolyah.ru/servera-i-protokoly/chto-takoe-klient-klientskij-kompyuter-i-klientskoe-prilozhenie.html) при помощи [HTTP сообщений](https://zametkinapolyah.ru/servera-i-protokoly/tema-3-http-soobshhenie-tipy-http-soobshhenij-zagolovki-http-soobshhenij-telo-soobshheniya-i-primery-http-soobshhenij.html). Поскольку в основе протокола лежит взаимодействие клиент-сервер, то предполагается, что есть клиент, который делает [HTTP запросы](https://zametkinapolyah.ru/servera-i-protokoly/tema-4-http-zapros-zagolovki-http-zaprosa-metody-http-zaprosa-stroka-http-zaprosa-resursy-http-zaprosa-primery-zaprosov.html) и есть [HTTP сервер](https://zametkinapolyah.ru/servera-i-protokoly/http-server-ili-veb-server-naznachenie-funkcii-i-rol-servera-v-http.html), который обрабатывает эти запросы и дает клиенту [HTTP ответы](https://zametkinapolyah.ru/servera-i-protokoly/tema-5-http-otvety-servera-stroka-sostoyaniya-http-otveta-kody-sostoyaniya-zagolovki-http-otveta-primery.html). Все ответы сервера содержат [коды состояния](https://zametkinapolyah.ru/servera-i-protokoly/spisok-kodov-sostoyaniya-http-servera-spravochnik-kodov-sostoyaniya-http-servera.html), а все запросы клиента имеют [HTTP методы](https://zametkinapolyah.ru/servera-i-protokoly/tema-7-opredelenie-metodov-http-http-method-definitions-metody-http-zaprosov.html).

Как мы уже поняли, HTTP следует модели запроса/ответа, когда клиент, подключенный к серверу, отправляет запрос, а сервер отвечает на него.

HTTP-сообщение (запрос или ответ) состоит из нескольких частей:

* «first line» (первая строка)
* headers (заголовки запроса)
* body (тело запроса)

### 1.2.1 Кратко о методе HTTP запроса

В запросе первая строка указывает метод, используемый клиентом, путь к ресурсу, который он хочет, а также версию протокола, который он собирается использовать:

GET /players/lebron-james HTTP/1.1

В этом случае клиент пытается получить ресурс (GET) по адресу /Players/Lebron-James через версию протокола 1.1 .

### 1.2.2 Заголовок (HTTP заголовки)

HTTP заголовки могут содержать описание данных и информацию необходимую для взаимодействия между клиентом и сервером. Заголовки и их статусы перечислены в [реестре IANA](http://www.iana.org/assignments/message-headers/perm-headers.html), который постоянно обновляется.

После первой строки HTTP позволяет нам добавлять метаданные к сообщению через заголовки, которые принимают форму ключ-значение, разделенных двоеточием:

GET /players/lebron-james HTTP/1.1

Host: nba.com

Accept: \*/\*

Coolness: 9000

Например, в этом запросе клиент добавил к запросу 3 дополнительных заголовка: Host, Accept и Coolness

Заголовки не должны использовать определенные, зарезервированные имена, но обычно рекомендуется полагаться на те, которые стандартизированы в спецификации HTTP: чем больше вы отклоняетесь от стандартов, тем меньше вас поймет другой участник обмена.

Cache-Control — это, например, заголовок, используемый для определения того, является ли (и каким образом) ответ кешируемым: большинство прокси и обратных прокси понимают его, следуя спецификации HTTP. Если бы вам пришлось переименовать заголовок Cache-Control в Awesome-Cache-Control, прокси не имели бы представления о том, как кэшировать ответ, так как они не созданы для соответствия спецификации, которую вы только что придумали.

Однако иногда имеет смысл включить в сообщение «пользовательский» заголовок, так как вы можете добавить метаданные, которые на самом деле не являются частью спецификации HTTP: сервер может решить включить техническую информацию в свой ответ, чтобы клиент мог одновременно выполнять запросы и получать важную информацию о состоянии сервера, который возвращает ответ:

...

X-Cpu-Usage: 40%

X-Memory-Available: 1%

...

При использовании пользовательских заголовков всегда предпочтительно ставить перед ними префикс с ключом, чтобы они не конфликтовали с другими заголовками, которые могут стать стандартом в будущем: исторически это работало хорошо, пока все не начали использовать «нестандартные» префиксы X что, в свою очередь, стало нормой. Заголовки X-Forwarded-For и X-Forwarded-Protoявляются примерами пользовательских заголовков, которые [широко используются и понимаются балансировщиками нагрузки и прокси-серверами](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers#Proxies), даже если они [не являются частью стандарта HTTP](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html).

Если вам нужно добавить собственный настраиваемый заголовок, в настоящее время обычно лучше использовать фирменный префикс, такой как Acme-Custom-Header или A-Custom-Header.

### 1.2.3 Тело

После заголовков запрос может содержать тело, которое отделено от заголовков пустой строкой:

POST /players/lebron-james/comments HTTP/1.1

Host: nba.com

Accept: \*/\*

Coolness: 9000

Best Player Ever

Наш запрос завершен: первая строка (информация о местоположении и протоколе), заголовки и тело. Обратите внимание, что тело является полностью необязательным и, в большинстве случаев, оно используется только тогда, когда мы хотим отправить данные на сервер, поэтому в приведенном выше примере используется метод POST.

### 1.2.4 Пример ответа

Ответ сильно не имеет различий:

HTTP/1.1 200 OK

Content-Type: application/json

Cache-Control: private, max-age=3600

{"name": "Lebron James", "birthplace": "Akron, Ohio", ...}

Первая информация, которая присылается в ответе, — это версия протокола, которую он использует, вместе со статусом этого ответа. Далее следуют заголовки и, если требуется, разрыв строки, за которым следует тело.

Как уже упоминалось, протокол подвергся многочисленным пересмотрам и со временем добавились новые функции (новые заголовки, коды состояния и т. д.), Но основная структура не сильно изменилась (первая строка, заголовки и тело). Что действительно изменилось, так это то, как клиенты и серверы обмениваются этими сообщениями.

### 1.2.5 Типы запроса (Методы запросов)

HTTP определяет множество методов запроса, которые указывают, какое желаемое действие выполнится для данного ресурса. Несмотря на то, что их названия могут быть существительными, эти методы запроса иногда называются HTTP глаголами. Каждый реализует свою семантику, но каждая группа команд разделяет общие свойства: так, методы могут быть [безопасными](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/safe), [идемпотентными](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/idempotent) или [кэшируемыми](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/cacheable).

[GET](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/GET)- Метод GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные;

[HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) - Метод HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа;

[POST](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/POST) - Метод POST используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере;

[PUT](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT) - Метод PUT заменяет все текущие представления ресурса данными запроса;

[DELETE](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE) - Метод DELETE удаляет указанный ресурс;

[CONNECT](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/CONNECT) - Метод CONNECT устанавливает "туннель" к серверу, определённому по ресурсу;

[OPTIONS](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/OPTIONS) - OPTIONS используется для описания параметров соединения с ресурсом;

[TRACE](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/TRACE) - Метод TRACE выполняет вызов возвращаемого тестового сообщения с ресурса;

[PATCH](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH) - Метод PATCH используется для частичного изменения ресурса.

### 1.2.5 HTTPS

HTTPS является «безопасным» расширением HTTP и включает установление общего секретного ключа между клиентом и сервером, гарантируя, что мы общаемся с нужной стороной, и шифруем сообщения, которые обмениваются общим секретным ключом. Пока HTTPS был нацелен на повышение безопасности протокола HTTP, H2 был нацелен на обеспечение высокой скорости.

HTTPS (HTTP Secure) позволяет клиентам и серверам безопасно общаться через TLS (Transport Layer Security), преемник SSL (Secure Socket Layer).

Проблема, на которую ориентирован TLS, довольно проста и может быть проиллюстрирована одной простой метафорой: ваша вторая половинка звонит вам в середине дня, когда вы находитесь на собрании, и просит вас сообщить им пароль учетной записи вашего онлайн-банкинга, поскольку она должна выполнить банковский перевод, чтобы обеспечить своевременную оплату за обучение вашего сына. Очень важно, чтобы вы сообщили об этом прямо сейчас, иначе вы столкнетесь с вероятностью того, что вашего ребенка отчислят из школы на следующее утро.

Теперь вы столкнулись с двумя проблемами:

Aутентификация (authentication) того, что вы действительно разговариваете со своей второй половинкой, поскольку это может быть кто-то, притворяющийся ей;

Шифрование (encryption): передача пароля так, чтобы ваши коллеги не смогли его понять и записать.

Что вы будете делать? Это именно та проблема, которую HTTPS пытается решить.

Чтобы проверить, с кем вы разговариваете, HTTPS использует Public Key Certificates(сертификаты открытых ключей), которые представляют собой не что иное, как сертификаты, указывающие личность конкретного сервера: когда вы подключаетесь через HTTPS к IP-адресу, сервер за этим адресом представляет вам его сертификат для вас, чтобы подтвердить свою личность. Возвращаясь к нашей аналогии, вы можете просто попросить свою вторую половинку сказать ваш номер социального страхования. Как только вы убедитесь, что номер правильный, вы получаете дополнительный уровень доверия.

Это, однако, не мешает «злоумышленникам» узнать номер социального страхования жертвы, украсть смартфон вашей второй половинки и позвонить вам. Как мы проверим личность звонящего?

Вместо того, чтобы прямо попросить свою вторую половинку написать свой номер социального страхования, вместо этого вы звоните своей маме (которая живет по соседству) и просите ее пойти в вашу квартиру и удостовериться, что именно ваша вторая половина говорит номер социального страхования. Это добавляет дополнительный уровень доверия, так как вы не считаете свою маму угрозой и полагаетесь на нее для проверки личности звонящего.

В терминах HTTPS ваша мама называется CA, сокращение от Certificate Authority: работа CA заключается в проверке личности конкретного сервера и выдаче сертификата с собственной цифровой подписью: это означает, что при подключении к определенному домену я получу не сертификат, сгенерированный владельцем домена (так называемый [самоподписанный сертификат](https://en.wikipedia.org/wiki/Self-signed_certificate)), а CA.

Задача CA состоит в том, что они проверяют подлинность домена и выдают сертификат соответствующим образом: когда вы «заказываете» сертификат (обычно называемый SSL-сертификатом, хотя в настоящее время вместо него используется TLS — названия действительно прилипают!), CA могут позвонить вам или попросить изменить настройку DNS, чтобы убедиться, что вы контролируете данный домен. После завершения процесса проверки он выдаст сертификат, который затем можно установить на веб-серверы.

Затем клиенты, такие как браузеры, будут подключаться к вашим серверам и получать этот сертификат, чтобы они могли проверить его подлинность: браузеры имеют своего рода «отношения» с CA, в том смысле, что они отслеживают список доверенных доменов в CA чтобы убедиться, что сертификат действительно заслуживает доверия. Если сертификат не подписан доверенным органом, браузер отобразит большое информационное предупреждение для пользователей:

Мы на полпути к обеспечению связи между вами и вашей второй половиной: теперь, когда мы прошли аутентификацию (проверку личности вызывающего абонента), нам нужно убедиться, что мы можем общаться безопасно, без вмешательства других в процесс. Как я уже упоминал, вы находитесь прямо посреди собрания и вам нужно записать свой пароль для онлайн-банкинга. Вам нужно найти способ зашифровать ваше общение, чтобы только вы и ваша родственная душа могли понять ваш разговор.

Вы можете использовать вариации [протокола обмена ключами Диффи-Хеллмана](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange), которые гарантируют, что стороны без предварительного знания устанавливают общий секретный ключ, и никто другой не сможет его «украсть». Это включает в себя [использование математики](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Cryptographic_explanation).

Как только секретный ключ установлен, клиент и сервер могут общаться, не опасаясь, что кто-то может перехватить их сообщения. Даже если злоумышленники сделают это, у них не будет общего секретного ключа, необходимого для расшифровки сообщений.

Для получения дополнительной информации о HTTPS и Diffie-Hellman я бы порекомендовал прочитать «[Как HTTPS защищает соединения](https://blog.hartleybrody.com/https-certificates/)» Хартли Броди и «[Как HTTPS на самом деле работает?](https://robertheaton.com/2014/03/27/how-does-https-actually-work/)» Роберта Хитона. Кроме того, в «[Девяти алгоритмах, которые изменили будущее](https://en.wikipedia.org/wiki/9_Algorithms_That_Changed_the_Future)» есть удивительная глава, в которой объясняется шифрование с открытым ключом, и я горячо рекомендую его фанатам компьютерных наук, интересующимся оригинальными алгоритмами.

### 1.2.6 Кратко о HTTP2 (H2)

H2 использует двоичные, а не текстовые сообщения, поддерживает мультиплексирование, использует алгоритм HPACK для сжатия заголовков…… Короче говоря, H2 повышает производительность по HTTP/1.1.

### 1.2.7 Актуальные проблемы работы с HTTP (нет ответа)

### 1.2.8 Простейшие инструменты для анализа HTTP запроса (нет ответа)

## 1.3. Нововведение в ASP .NET MVC 5

Аутентификация и авторизация;

Создание адаптивного и расширяемого интерфейса с использованием css-фреймворка Bootstrap;

Добавлены фильтры аутентификации, а также появилась функциональность переопределения фильтров;

Добавлены атрибуты маршрутизации;

Использование по умолчанию Entity Framework 6.

## 1.4. Предназначение ASP .NET MVC 5

Платформа ASP.NET MVC представляет собой фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна MVC.

## 1.5 Обзор Asp.Net

ASP.NET — это веб-платформа, предоставляющая все необходимые службы для создания серверных веб-приложений корпоративного класса с помощью .NET в Windows.

Также можно создавать веб-API и использовать технологии реального времени, такие как веб-сокеты.

ASP.NET предлагает три платформы для создания веб-приложений: Веб-формы, ASP.NET MVC и веб-страницы ASP.NET.

[ASP.NET Core](https://docs.microsoft.com/aspnet/core/) является альтернативой ASP.NET. Смотрите [рекомендации по выбору между ASP.NET и ASP.NET Core](https://docs.microsoft.com/aspnet/core/choose-aspnet-framework).

### 1.5.1 Web Pages Applications. Веб-страницы ASP.NET

Веб-страницы ASP.NET и синтаксис Razor обеспечивают быстрый, понятный и простой способ объединения серверного кода с HTML для создания динамического веб-содержимого. Подключайтесь к базам данных, добавляйте видео, ссылки на сайты социальных сетей и множество других дополнительных функций, чтобы создавать сайты, которые соответствуют новейшим веб-стандартам.

[Дополнительные сведения о веб-страницах](https://docs.microsoft.com/ru-ru/aspnet/web-pages/index)

### 1.5.2 Web Forms Applications. Веб-формы

С веб-формами ASP.NET можно создавать динамические веб-сайты, используя знакомую модель перетаскивания, управляемую событиями. Область конструирования и сотни элементов управления и компонентов позволяют быстро создавать комплексные сайты с эффективным пользовательским интерфейсом и доступом к данным. [Дополнительные сведения о веб-формах](https://docs.microsoft.com/ru-ru/aspnet/web-forms/index)

### 1.5.3 MVC Applications.

ASP.NET MVC предлагает эффективный, основанный на шаблонах способ создания динамических веб-сайтов, который позволяет четко разделять проблемы и дает полный контроль над разметкой для гибких разработок. ASP.NET MVC содержит множество функций, позволяющих вести быструю TDD-совместимую разработку для создания сложных приложений, использующих новейшие веб-стандарты. [Дополнительные сведения о MVC](https://docs.microsoft.com/ru-ru/aspnet/mvc/index)

### 1.5.4 Преимущества MVC приложения

#### 1.5.4.1 Расширяемость

Для каждого компонента MVC Framework разработчику представляется три возможности:

Использование стандартного компонента;

Порождение подкласса от стандартной реализации;

Полная замена компонента новой реализацией интерфейса.

#### 1.5.4.2 Жесткий контроль над HTML и HTTP

Генерация разметки, которая соответствет стандартам;

Отсутствие ViewState;

Полный контроль над запросами.

#### 1.5.4.3 Тестируемость

Разнесение различных задач приложения по разным, независимых друг от друга частям программного обеспечения, позволяет строить легко тестируемое приложение.

#### 1.5.4.4 Мощная система маршрутизации (нет ответа)

## 1.6 Общие возможности для Asp.Net приложений (нет ответа)

<https://professorweb.ru/my/ASP_NET/base/level1/1_1.php>

### 1.6.1 конфигурация (нет ответа)

### 1.6.2 аутентификация (нет ответа)

### 1.6.3 роли и провайдеры (нет ответа)

### 1.6.4 управление состоянием (нет ответа)

### 1.6.5 кэширование (нет ответа)

## 1.7 Введение в MVC приложение

ASP.NET MVC - это фреймворк для разработки веб-приложений от Microsoft, которая сочетает в себе эффективность и аккуратность архитектуры "модель-представление-контроллер" (model-view-controller - MVC), новейшие идеи и приемы гибкой разработки, а также все лучшее из существующей платформы ASP.NET. Она представляет собой полномасштабную альтернативу традиционной технологии ASP.NET Web Forms, предоставляя преимущества для всех проектов веб-разработки, кроме самых тривиальных.

Начнем с главного – Model-View-Controller. MVC — это фундаментальный паттерн, согласно которому веб-приложение делится на компоненты Model View Controller. Впервые паттерн MVC появился в языке SmallTalk. Разработчики должны были придумать архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнес логики, а бизнес логику от данных. Таким образом, в классическом варианте, MVC состоит из трех частей, которые и дали ему название. Рассмотрим их: Модель, представление, контроллер.

### 1.7.1 Модель, представление, контроллер. Понимание паттерна MVC.

#### 1.7.1.1 Модель (Model)

Под Моделью, обычно понимается элемент приложения содержащий в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

Модель обладает следующими признаками:

* Модель — это бизнес-логика приложения;
* Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях;
* Для некоторых проектов модель — это просто слой данных (DAO, база данных, XML-файл);
* Для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения;

Обычно различают несколько типов паттернов в зависимости от роли модели.

Passive Model (пассивная модель) - Модель не имеет вообще никаких способов воздействовать на Представление или Контроллер и только используется ими в качестве источника данных для отображения. Все изменения модели отслеживаются Контроллером и он же отвечает за перерисовку Представления, если это необходимо.

Active Model (активная модель) - Модель имеет возможность оповестить Представление о том, что в ней произошли некие изменения, и Представление может эти изменения отобразить. Как правило, механизм оповещения реализуется на основе паттерна Observer (обозреватель), Модель просто бросает сообщение, а Представления, которые заинтересованы в оповещении, подписываются на эти сообщения, что позволяет сохранить независимость Модели как от Контроллера так и от Представления, не нарушая тем самым основного свойства паттерна. Классической реализацией паттерна MVC принято считать версию именно с активной Моделью.

#### 1.7.1.2 Представление (View)

В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным. В случае активной Модели, Представление может подписаться на события изменения Модели и перерисовываться, забрав измененные данные, при получении соответствующего оповещения. Для взаимодействия с Контроллером, представление, как правило, реализует некий интерфейс, известный Контроллеру, что позволяет менять представления независимо и иметь несколько представлений на Контроллер. Вообще, подмена или изменение Представления самая часто встречающаяся задача, по сути это и есть та причина по которой придумывают различные паттерны разделения Модели и Представления.

Представление обладает следующими признаками:

* В представлении реализуется отображение данных, которые получаются от модели любым способом;
* В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

В MVC представление представлено в \*.cshtml файлами.  
 Чтобы построить хорошее представление надо знать HTML, CS, JS.

#### 1.7.1.3 Контроллер (Controller)

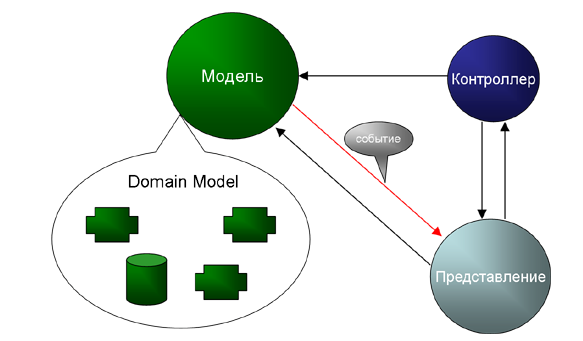
Контроллер обрабатывает поступающие HTTP запросы, выполняет операции с моделью и выбор представления для визуализации интерфейса.

Контроллер обеспечивает «связь» между пользователем и системой. Контролирует и направляет данные от пользователя к системе и наоборот. Использует модель и представление для реализации необходимого действия.

В задачи Контроллера входит реакция на внешние раздражители и изменение Модели и/или Представления в соответствии с заложенной в него логикой. Один Контроллер может работать с несколькими Представлениями, в зависимости от ситуации, взаимодействуя с ними через некий заранее известный интерфейс, который эти Представления реализуют. Важный нюанс, в классической версии MVC Контроллер не занимается передачей данных из Модели в Представление и не является медиатором (Mediator) между Моделью и Представлениями.

#### 1.7.1.4 Понимание паттерна MVC.

Таким образом, типичная схема взаимодействия компонентов паттерна выглядит примерно так: Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой, реагирует на это событие изменяя Mодель, посредством вызова соответствующего метода Модели. После изменения Модель бросает событие о том что она изменилась, и все подписанные на это события Представления, получив его, обращаются к Модели за обновленными данными, после чего их и отображают.

При этом еще в описании оригинального паттерна упоминалось, что выделение отдельного Контроллера не так важно как отделение Представления от Модели и Контроллер вполне может быть интегрирован в Представление.

Следующим этапом развития MVC стал паттерн Document-View, хорошо известный по таким библиотекам как Turbo Vision (Pascal 6.0, Microsoft Foundation Class Library и многих других, вплоть до WinForms.

|  |
| --- |
| ПРИМЕЧАНИЕ  Строго говоря, утверждать, что эти библиотеки реализуют именно Document-View было бы не совсем верно, в конце-концов это всего лишь библиотеки и использовать их можно по разному, вернее было бы сказать, что эти библиотеки склоняют к использованию MVC именно в таком виде, как примерами из документации, так и собственной архитектурой. |

В этой версии MVC Контроллер интегрирован в Представление, что ни в коей мере не является нарушением основной идеи паттерна. Сделано это было по многим причинам, прежде всего, отделение Контроллера от Представления, действительно не самая ключевая часть паттерна. Другой причиной являлось появление, графических оболочек встроеных в ОС, что позволяло не рисовать графические элементы (контролы) пользовательского интерфейса под каждый проект, а использовать готовые, предоставляемые платформой посредством соответствующего API, но дело в том, что в этих оболочках функции Контроллера уже были интегрированы в контролы (которые и являются Представлениями или же частями оного). Свою роль сыграло и появление визуальных графических оболочек встроеных в среду программирования (так называемые widget-based среды пользовательского интерфейса), поскольку код сгенеренный этими оболочками, естественно, использовал готовые графические элементы платформы и, как следствие, опять-таки спровоцировал разработку в стиле Document-View. WinForms, в купе с Visual Studio, также являются примером такой среды.

У той же Microsoft, которая выкладывает довольно много статей по архитектурным решениям в свободный доступ, есть несколько публикаций и по MVC, и там можно легко заметить, что все примеры чистого MVC даются не на основе WinForms приложений, а на примере ASP.NET, где, в отличии от WinForms, есть относительно четкое разделение между Контроллером и Представлением, про WinForms же есть только упоминание о том, что, как правило, в подобного рода приложениях Контроллер интегрирован в Представление.

### 1.7.2 Различия MVC & MVVM & MVP

Наиболее распространенные виды MVC-паттерна, это: MVC, MVP и MVVM рассмотрим и сравним каждый из них.

#### 1.7.2.1 Model-View-Presenter

Данный подход позволяет создавать абстракцию представления. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу изменяет модель.

Признаки презентера:

- Двусторонняя коммуникация с представлением;

- Представление взаимодействует напрямую с презентером, путем вызова соответствующих функций или событий экземпляра презентера;

- Презентер взаимодействует с View путем использования специального интерфейса, реализованного представлением;

- Один экземпляр презентера связан с одним отображением.

Реализация:

Каждое представление должно реализовывать соответствующий интерфейс. Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем (например, IView.ShowErrorMessage(string msg)). Презентер должен иметь ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе.

Логика представления должна иметь ссылку на экземпляр презентера. Все события представления передаются для обработки в презентер и практически никогда не обрабатываются логикой представления (в т.ч. создания других представлений).

Пример использования: Windows Forms.

#### 1.7.2.2 Model-View-View Model



Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

Признаки View-модели:

- Двусторонняя коммуникация с представлением;

- View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели / модели

- View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings)

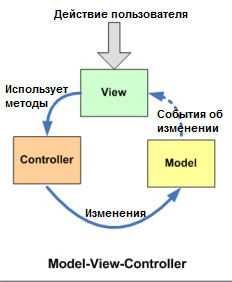
- Один экземпляр View-модели связан с одним отображением.

Реализация:

При использовании этого паттерна, представление не реализует соответствующий интерфейс (IView). Представление должно иметь ссылку на источник данных (DataContex), которым в данном случае является View-модель. Элементы представления связаны (Bind) с соответствующими свойствами и событиями View-модели. В свою очередь, View-модель реализует специальный интерфейс, который используется для автоматического обновления элементов представления. Примером такого интерфейса в WPF может быть INotifyPropertyChanged.

Пример использования: WPF

#### 1.7.2.3 Model-View-Controller



Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонентов.

Признаки контроллера

- Контроллер определяет, какие представление должно быть отображено в данный момент;

- События представления могут повлиять только на контроллер.контроллер может повлиять на модель и определить другое представление.

- Возможно несколько представлений только для одного контроллера;

Реализация:

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой, реагирует на это событие изменяя Mодель, посредством вызова соответствующего метода. После изменения Модель использует событие о том что она изменилась, и все подписанные на это события Представления, получив его, обращаются к Модели за обновленными данными, после чего их и отображают.

Пример использования: MVC ASP.NET

### 1.7.3 Общие правила выбора паттерна

#### 1.7.3.1 MVVM

* Используется в ситуации, когда возможно связывание данных без необходимости ввода специальных интерфейсов представления (т.е. отсутствует необходимость реализовывать IView);
* Частым примером является технология WPF.

#### 1.7.3.2 MVP

* Используется в ситуации, когда невозможно связывание данных (нельзя использовать Binding);
* Частым примером может быть использование Windows Forms.

#### 1.7.3.3 MVC

* Используется в ситуации, когда связь между представление и другими частями приложения невозможна (и Вы не можете использовать MVVM или MVP);
* Частым примером использования может служить ASP.NET MVC.

### 1.7.4 Обзор новых возможностей в MVC разработке.( нет ответа)

## 1.8 Хостинг Web приложений. Знакомство с IIS и IIS Express.

Хостинг сайтов или Web приложений – это онлайн услуга, которая позволяет публиковать ваш веб-сайт или веб-приложение в интернете. Когда вы подписываетесь на услугу хостинга, вы обычно арендуете пространство на сервере, на котором вы можете хранить все файлы и данные, необходимые для правильного функционирования вашего сайта.

Сервер – это физический компьютер, который работает без перерывов, чтобы ваш сайт был доступен всё время для тех, кто хочет его посетить. Ваш хостинг отвечает за поддержание работы сервера, защиту его от вредоносных атак и передачу вашего контента (текста, изображений, файлов) с сервера в браузеры ваших посетителей.

### 1.8.1 Как работает хостинг сайтов?

Когда вы решите запустить новый сайт, вам понадобиться найти хостинговую компанию, которая предоставит вам ресурсы на сервере. Ваш хостинг провайдер хранит все ваши файлы, ресурсы и базы данных на сервере. Всякий раз, когда кто-то вводит ваше доменное имя в адресную строку своего браузера, ваш хост передаёт все файлы, необходимые для обслуживания запроса.

Помимо предоставления серверного пространства для вашего сайта, хостинг провайдеры могут также предлагать другие услуги, связанные с управлением сайтов, такие как:

* SSL-сертификаты (для обеспечения безопасности сайтов используется протокол https://)
* Хостинг электронной почты Email
* Конструкторы страниц
* Инструменты для разработчиков
* Услуги поддержки клиентов (обычно с онлайн чатом)
* Автоматизированное создание бэкапов (резервных копий данных)
* Установщики программ в 1-клик мыши (например, CMS WordPress или Drupal)

### 1.8.2 Типы хостинга

Большинство провайдеров предлагают несколько типов хостинга для удовлетворения различных потребностей клиентов. Вот наиболее часто предоставляемые типы хостинга:

#### 1.8.2.1 Общий хостинг (Shared Hosting)

Общий хостинг или виртуальным хостингом является наиболее распространённым типом веб-хостинга, и это отличное решение для большинства небольших проектов и блогов. Общий хостинг предполагает, что вы совместно с другими клиентами вашего хостинга пользуетесь ресурсами сервера. Веб-сайты, расположенные на одном сервере разделяют свои ресурсы, такие как память, вычислительная мощность, дисковое пространство и другие.

Достоинства: Низкая стоимость, Удобно для начинающих (не требует специальных технических знаний), Настроенный сервер, Понятная панель управления, Обслуживание и администрирование сервера выполняются службой поддержки

Недостатки: Ограниченная возможность конфигурации (настройки) сервера, Потоки трафика на других сайтах могут замедлить работу вашего сайта

#### 1.8.2.2 [VPS хостинг](https://www.hostinger.ru/vps-hosting-servera) (Virtual Private Server – виртуальный приватный сервер)

С хостингом VPS (Virtual Private Server – виртуальный приватный сервер) вы по-прежнему используете сервер совместно с другими пользователями, однако ваш провайдер выделяет вам отдельный раздел на сервере. Это означает, что вы получаете выделенное пространство на сервере и зарезервированное количество вычислительной мощности и памяти. Фактически, VPS-хостинг может быть отличным для проектов среднего бизнеса с быстро растущими сайтами.

Достоинства: Выделенные ресурсы на сервере (без платы за выделенный сервер), Потоки трафика на другие сайты никак не влияют на производительность вашего сайта, Root-доступ к сервера, Лёгкая масштабируемость, Высокая настраиваемость

Недостатки: Дороже общего хостинга, Необходимы технические знания и знания по управлению серверами

#### 1.8.2.3 Облачный хостинг (Cloud Hosting)

Облачный хостинг в настоящее время является самым надёжным решением на рынке, поскольку он работает буквально бесперебойно. Облачный хостинг предоставляет вам кластер серверов. Ваши файлы и ресурсы реплицируются (копируются) на каждом сервере. Когда один из облачных серверов занят или имеет какие-либо проблемы, ваш трафик автоматически направляется на другой сервер в кластере.

Достоинства: Краткий или отсутствующий простой, Сбои на сервере не сказываются на работе вашего сайта, Ресурсы выделяются по требованию, Оплата по мере использования (платите только за то, что используете), Более масштабируемый, чем VPS

Недостатки: Сложно заранее просчитать стоимость, Не всегда предоставляется Root-доступ

#### 1.8.2.4 WordPress хостинг

WordPress Хостинг – это разновидность общего хостинга, специально созданная для размещения сайтов на WordPress. Ваш сервер настроен определённым образом наиболее подходящим для потребностей работы CMS WordPress, на вашем сайте сразу готовые заранее установленный плагины для таких важных моментов как, например, кэширование и безопасность. Благодаря высоко оптимизированной конфигурации ваш сайт загружается намного быстрее и работает с меньшим количеством проблем.

Достоинства: Низкая цена (обычно доступны по такой же цене, как и общий хостинг), Удобный для начинающих, Установка WordPress в одно нажатие, Отличная производительность для сайтов на WordPress, Служба поддержки клиентов хорошо разбирается в вопросах, связанных с WordPress, Заранее установленные плагины и темы для WordPress

Недостатки: Рекомендуется использовать только для сайтов на WordPress (может возникнуть проблема, если вы захотите разместить более одного сайта на своём аккаунте, и не все из них используют WordPress)

#### 1.8.2.5 Выделенный хостинг

Выделенный хостинг означает, что у вас есть собственный физический сервер, который предназначен исключительно для вашего сайта. Таким образом, выделенный хостинг даёт вам невероятную гибкость. Вы можете настроить свой сервер по своему усмотрению, выбрать операционную систему и программное обеспечение, которые хотите использовать, и настроить всю среду размещения в соответствии с вашими потребностями. Фактически, аренда выделенного сервера равносильна своему собственному локальному серверу, но поставляется с профессиональной поддержкой вашего хостинг провайдера.

Достоинства: Полное управление конфигурацией вашего сервера, Высокая надёжность (вы ни с кем не делите ресурсы своего сервера), Полный Root-доступ, Высокая безопасность

Недостатки: Высокая стоимость, Необходимы технические знания и знания по управлению серверами

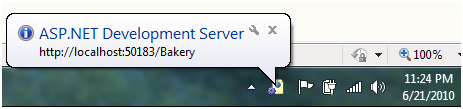
#### 1.8.2.6 Колокация «[Colocation](https://ru.wikipedia.org/w/index.php?title=Colocation&redirect=no)»

Колокация, колокейшн (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) co-location, сокращенно colo) — услуга, состоящая в том, что [провайдер](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82-%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80) размещает оборудование клиента на своей территории (обычно в [датацентре](https://ru.wikipedia.org/wiki/%D0%94%D0%B0%D1%82%D0%B0%D1%86%D0%B5%D0%BD%D1%82%D1%80)), подключает его к [электричеству](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C), обеспечивает обслуживание и подключение к каналам связи с высокой пропускной способностью.

### 1.8.3 Знакомство с IIS, IIS Express и ASP.NET Development Server

#### 1.8.3.1 ASP.NET Development Server

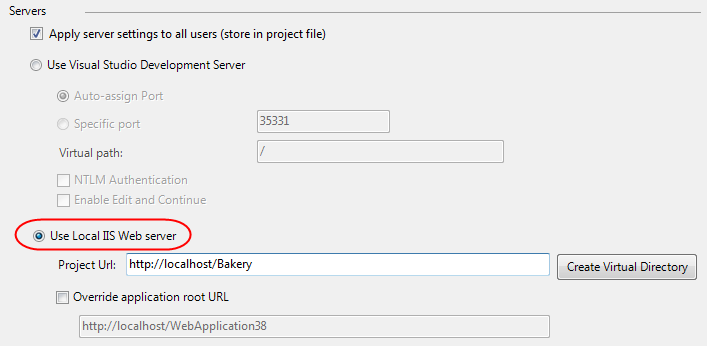
Встроенный в Visual Studio ASP.NET Development Server (также известен, как “Cassini”) обладает преимуществами легкого и быстрого запуска. Он не слушает удаленные порты, что делает проще работу в корпоративной среде в плане безопасности, работает даже когда у вас нет прав администратора, не требует отдельной установки на машину.



Недостатком же является неполный набор возможностей веб-сервера. Например, отсутствует поддержка SSL, URL Rewriting Rules (о котором я [недавно писал](http://weblogs.asp.net/scottgu/archive/2010/04/20/tip-trick-fix-common-seo-problems-using-the-url-rewrite-extension.aspx)), собственных настроек безопасности и другие возможности IIS 7.

#### 1.8.3.2 IIS веб-сервер

IIS (Internet Information Services) — альтернативный способ для запуска и тестирования приложений с Visual Studio. Вы можете настроить в Visual Studio любой веб-проект использовать IIS, нажав правой кнопкой по проекту, зайдя в свойства и далее выбрав закладку “Web”:



Используя IIS, как сервер для разработок вы получаете все возможности веб-сервера (SSL, URL Rewrite Rules и т.п.). IIS является полноценным веб-сервером, а это значит, что вы точно видите, как будет работать ваше приложения на публичном сервере.

IIS также требует прав администратора для установки и отладки проектов. Различные версии Windows поддерживают различные версии IIS, Например, если у вас Windows XP, то у вас будет стоять IIS 5.1, который не поддерживает все новые возможности IIS 7.x.

1.8.3.3 IIS Express – все лучшее

IIS Express - он объединяет в себе легкость использования ASP.NET Web Server и всю мощь IIS. Вот перечень особенностей:

* Он легче и прост в установке (менее 10 Mb для загрузки и супер-быстрая установка)
* Не требует прав администратора для запуска/отладки приложений из Visual Studio
* Предоставляет полный набор возможностей веб-сервера, включая SSL, URL Rewrite, поддержка медиа и других IIS 7.x модулей
* Поддерживает ту же модель расширения web.config, что и IIS 7.x
* Может быть установлен рядом с IIS и ASP.NET Development Server, не конфликтует абсолютно
* Работает на Windows XP и выше, дает все преимущества IIS 7.x на всех ОС-платформах

На IIS Express (как и на ASP.NET Development Server) можно быстро запустить любой сайт из директории на диске. Не требует никаких шагов регистрации или настройки.

[Оригинал](https://habr.com/ru/post/97758/)

## 1.9 Жизненный цикл запроса к серверу.

[Жизненный цикл HTTP запроса к серверу (видео) минимально и не раскрывает этот вопрос, но можно посмотреть](https://deworker.pro/edu/series/http-basics/http-lifecycle)

### 1.9.1 Работа конвейера веб-сервера IIS

Ключевую роль при обработке запросов к приложениям ASP.NET играет веб-сервер IIS. IIS прослушивает входящие запросы, определяет, как они должны обрабатываться, и отправляет пользователю определенный ответ.

IIS определяет число запросов, которые могут обрабатываться единовременно на одном ядре процессора компьютера. Если количество входящих запросов превышает это число, то все дополнительные запросы помещаются в очередь ожидания, которая также имеет свой лимит. Если же и данная очередь уже заполнена, то входящий запрос отвергается, а браузеру в ответ посылается статусный код ошибки 503.

Познакомимся с отдельными компонентами веб-сервера IIS.

HTTP.sys - драйвер уровня ядра HTTP.sys является посредником между приложением и операционной системой. В архитектуре IIS HTTP.sys является протокольным слушателем (protocol listeners). То есть HTTP.sys прослушивает все запросы, которые, используют протоколы HTTP и HTTPS, и затем передает эти запросы другим компонентам IIS для дальнейшей обработки.

HTTP.sys также представляет стек протокола HTTP. А именно HTTP.sys выполняет кэширование, постановку запросов в очередь, предобработку и ряд других функций.

Порт завершения ввода-вывода

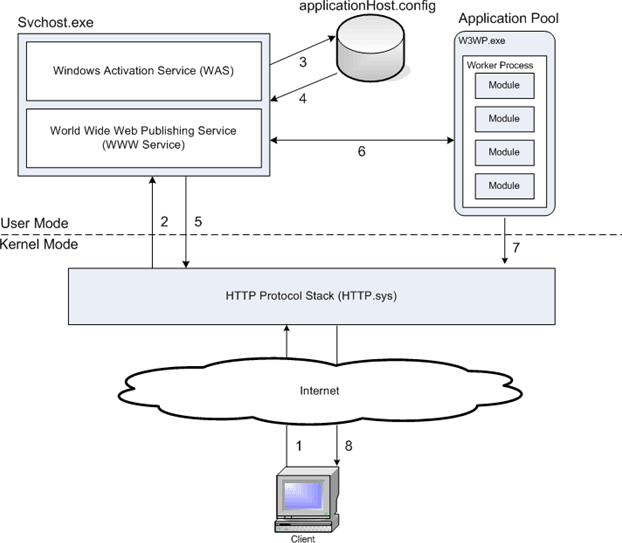
Для максимизации числа одновременно обрабатываемых запросов IIS использует специальное API, которое называется портом завершения ввода-вывода (Input/Output Completion Port). IIS имеет выделенный пул потоков для обработки операций ввода/вывода и управления очередью запросов. При обработке запросов этот API взаимодействует с драйвером HTTP.sys.

World Wide Web Publishing Service (W3SVC) или Служба веб-публикации является адаптером над драйвером HTTP.sys и выполняет функцию конфигурирования HTTP.sys и мониторинга показателей производительности

Windows Process Activation Service (WAS) данная служба была выделена начиная с IIS 7.0. Ее предназначение - управление рабочими процессами. WAS состоит из трех основных компонентов:

* менеджер конфигурации: считывает конфигурационную информацию о веб-приложениях и пулах приложений из файла applicationhost.config
* менеджер процессов: выполняет сопоставление между пулами приложений и рабочими процессами. Если соответствующий пулу приложения рабочий процесс еще не запущен, то менеджер процессов его запускает.
* интерфейс неуправляемых адаптеров прослушивателей (unmanaged listener adaptor interface): предоставляет интерфейс для прослушивателей запросов, которые не относятся к протоколу HTTP

#### 1.9.1.1 Обработка запросов в IIS



Теперь рассмотрим поэтапно работу конвейера IIS по обработке входящих запросов.

1. Пользователь посылает HTTP-запрос, обращаясь через браузер к определенному ресурсу на сервере. Этот запрос перехватывается драйвером HTTP.sys.
2. Драйвер HTTP.sys обращается к WAS для получения конфигурационных данных для запрошенного адреса URL
3. Менеджер конфигурации WAS считывает данные из файла applicationhost.config, в частности пул приложения и конфигурационные настройки сайта
4. Считанные данные WAS передает службе W3SVC
5. Служба веб-публикации (W3SVC) использует полученные от WAS данные для конфигурации HTTP.sys. Затем драйвер HTTP.sys помещает пришедший запрос в очередь порта завершения ввода-вывода (I/O completion port), которую обрабатывает WAS.
6. WAS использует выделенный пул потоков для обработки очереди. Если к данному моменту не был запущен рабочий процесс, ассоциированный с запрошенным URL, то WAS запускает приложение w3wp.exe, в рамках которого работает рабочий процесс обработки запросов.
7. В рамках запущенного рабочего процесса ASP.NET проверяет, сколько запросов обрабатывается в текущий момент времени. Если их число превышает лимит, то новый запрос помещается в очередь. Однако если очередь достигла своего лимита, то данный запрос отвергается, и в ответ клиенту посылается статусный код ошибки 503.  
   Если запрос направлен к статическому файлу, который не содержит кода .NET-языков, то ASP.NET посылает содержимое этого файла в порт завершения ввода-вывода IIS, а оттуда - пользователю, сделавшему запрос. В остальных случаях ASP.NET отправляет запрос в пул потоков CLR.  
   CLR обрабатывает запрос. На этой стадии запускается непосредственно код нашего веб-приложения или веб-сайта, размещенного на веб-сервере.  
   После того, как среда CLR закончит обработку запроса, она посылает результаты драйверу HTTP.sys, а тот - на порт завершения ввода-вывода IIS.
8. Пользователь получает результат обработки

### 1.9.2 Основы архитектуры IIS, или запросопровод для ASP.NET

статья ориентирована в большей степени на IIS 7.x, иногда будут ответвления про 6-ку.

#### 1.9.2.1 Общий план

В англоязычной литературе процесс обработки запроса в IIS называется «request processing pipeline» — что-то вроде «конвейера обработки запроса». В общих чертах он представлен на рисунке ниже для http-запроса.

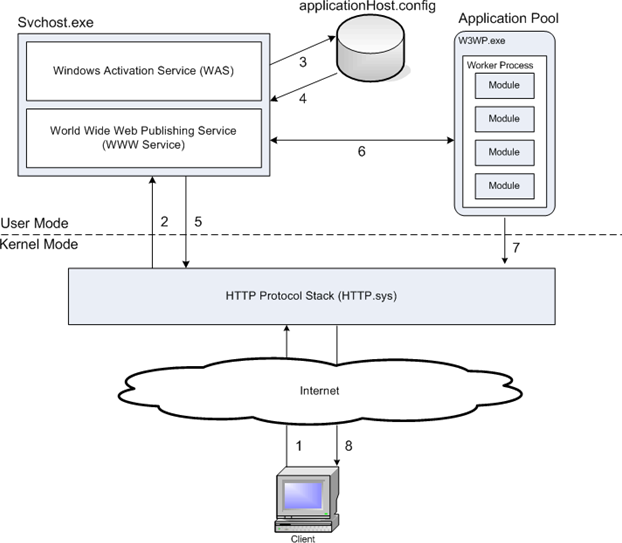


Рис. 1. HTTP request processing pipeline (IIS 7.x).

Таким образом, http-запрос проходит по «сборочной ленте конвейера» через следующее:

1. Браузер обращается к веб-серверу по определённому URL, на стороне сервера запрос перехватывает драйвер HTTP.SYS.

2. HTTP.SYS стучится к WAS для получения информации из хранилища конфигурации.

3. Служба WAS запрашивает конфигурацию из хранилища — из файла в папке IIS (applicationHost.config).

4. Поскольку данный запрос получен по протоколу HTTP конфигурационную информацию получает служба W3SVC (она же WWW Service на картинке), эта информация содержит в себе данные о пуле приложений (application pool) и прочих параметрах сайта.

5. Служба W3SVC использует эту информацию для конфигурации HTTP.SYS.

6. Служба WAS запускает процесс W3WP.exe для пула приложений, если он ещё не был запущен.

7. В процессе W3WP.exe работает приложение веб-сайта, которое, собственно, формирует и возвращает ответ драйверу HTTP.SYS.

8. HTTP.SYS отправляет ответ браузеру.

#### 1.9.2.2 Крупный план

HTTP.SYS

На транспортном уровне IIS использует protocol listeners, которые располагаются поверх стека TCP/IP. Наиболее интересный нам такой компонент – это системный драйвер HTTP.sys, который встроен в ядро ОС и работает с протоколами HTTP и HTTPS, нацелен на прослушку всех портов, на которые будут приходить запросы к сайтам в IIS.

Драйвер принимает все входящие запросы и перенаправляет их в нужный пул приложений. Если по какой-то причине рабочий процесс, в коем хостится требуемый пул, остановлен (сбой, таймаут простоя, смена конфигурации и т.п.) или ещё запускается, то HTTP.sys сохраняет входящие запросы в специально отведенной для каждого пула очереди.

Ещё HTTP.sys умеет [кешировать ответы](https://support.microsoft.com/en-us/help/817445/instances-in-which-http-sys-does-not-cache-content), поэтому некоторые запросы обрабатываются без передачи на уровень приложения, а также проводит первичный разбор URI запроса и его валидацию в соответствии с [RFC 2396](https://blogs.iis.net/nazim/use-of-special-characters-like-in-an-iis-url) и журналирование запросов/ответов.

Некоторые настройки HTTP.sys вынесены в системный [реестр Windows](https://support.microsoft.com/en-us/help/820129/http-sys-registry-settings-for-windows). Там же – в реестре – можно подсмотреть обычное место прописки нашего гражданина: %SystemRoot%\system32\drivers\http.sys..

2.2. World Wide Web Publishing Service (W3SVC)

В IIS 7.x функция управления процессами была вынесена в отдельную службу – WAS (см. п.2.3) в целях универсализации архитектуры. Теперь WWW-служба стала по своей сути одним из адаптеров, специализируясь на протоколах HTTP/HTTPS – работа поверх драйвера HTTP.sys. Однако WWW-служба остаётся краеугольным компонентом IIS, поэтому её настройка отличается от настройки адаптеров к другим протоколам; она функционирует в том же рабочем процессе, что и WAS, и реализована в той же самой библиотеке (рис. 2).

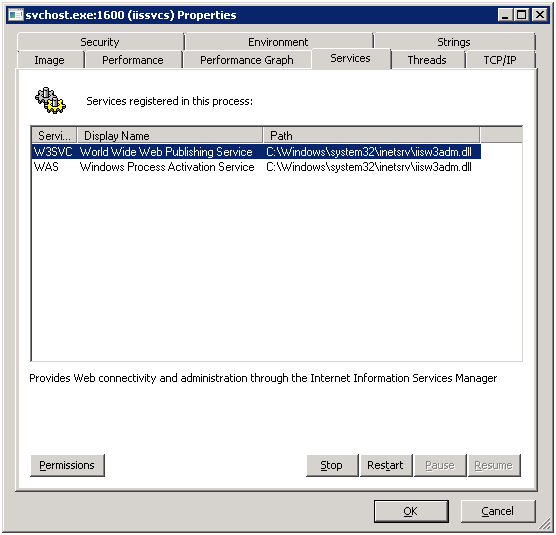


Рис.2. Рабочий процесс со службами W3SVC и WAS.

Раз уж зашла речь об адаптерах к прослушивателям протоколов (protocol listener adpater), то давайте чуть задержимся и посмотрим, какие они бывают. В принципе IIS 7.x можно настроить для обработки запросов по любым протоколам помимо типовых HTTP и FTP, например, POP3, SMTP, Gopher. Вы даже вольны придумать свой протокол для своей веб- или WCF-службы. Но прежде всего стоить обратить внимание на стандартные службы (рис. 3), поставляемые с .NET Framework и интегрированные с IIS:

* NetTcpActivator для протокола TCP;
* NetPipeActivator для Named Pipes;
* NetMsmqActivator для Message Queuing (ака MSMQ).

Но всё-таки наиболее важным для нас адаптером является именно WWW-служба, т.ч. остановимся чуть подробнее на двух оставшихся от IIS 6 функциях.

Администрирование и конфигурирование HTTP(S). В момент обновления конфигурации веб-сайтов, служба WAS передаёт эту информацию WWW-службе, а та уже, в свою очередь, настраивает HTTP.sys на прослушку конкретных портов, разбор IP и заголовка запрашиваемого сайта и, возможно, других параметров драйвера. В обратную сторону W3SVC обращается к WAS, когда в очередь запросов в HTTP.sys поступает новый, – для получения рабочего процесса-обработчика данного запроса.

Отслеживание показателей производительности. WWW-служба ведёт счётчики производительности, используя для этого драйвер HTTP.sys, и предоставляет их показатели веб-сайтами и кэшу IIS.

2.3. Windows Process Activation Service (WAS)

Итак, WWW-служба в IIS 7.x, как и в IIS 6, продолжает выполнять задачи по администрированию HTTP.sys и управлению показателями производительности веб-сайтов. А вот задача управления рабочими процессами вынесена в отдельную службу – WAS. Она запускается системой в единственном экземпляре, считывает конфигурацию из файла %\Config\ApplicationHost.config и настраивает через соответствующие адаптеры прослушивателей протоколов в соответствии с указанной в нём информации. Напомним, что для протоколов HTTP/HTTPS адаптером является служба W3SVC, а прослушивателем – драйвер HTTP.sys. При перехвате прослушивателем запроса он через свой адаптер обращается к службе WAS для получения рабочего процесса приложения, которому будет передан запрос для обработки и формирования ответа клиенту.

При активации нужного для обработки пользовательского запроса приложения задействуются следующие компоненты:

* Адаптеры прослушивателей (Listener adapters) – специальные службы Windows, работающие с конкретным протоколом и взаимодействующие с WAS для направления запросов к правильному рабочему процессу.
* Собственно WAS. Она ответственна за создание рабочих процессов и управление их временем жизни.
* Исполняемый файл w3wp.exe – шаблон рабочего процесса.
* Менеджер приложений управляет созданием и утилизацией доменов приложений (application domains), которые хостятся внутри рабочего процесса.
* Обработчики протоколов – протокол зависимые компоненты внутри рабочего процесса, ответственные за обмен данными между конкретным адаптером и рабочим процессом. Есть 2 типа обработчиков протоколов: у процесса (process protocol handler — PPH) и у домена приложения (AppDomain protocol handlers — ADPH).

Ниже на рисунке представлен пример схемы компонентов внутри некоего экземпляра рабочего процесса приложения. Когда служба WAS запускает рабочий процесс, она загружает в него согласно конфигурации приложения требуемые обработчики протоколов процессов (PPH) и посредством менеджера приложений создаёт внутри рабочего процесса домен приложения, в котором будет хоститься приложение. Менеджер приложений загружает код приложения в домен приложения и требуемые обработчики протоколов уровня приложения (ADPH) для обработки сообщений по соответствующим сетевым протоколам.

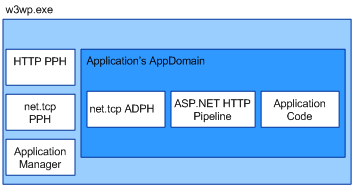
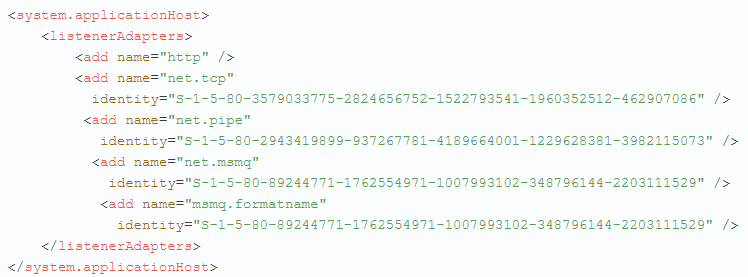


Рис. 4. Компоненты w3wp.exe для взаимодействия с внешними компонентами.

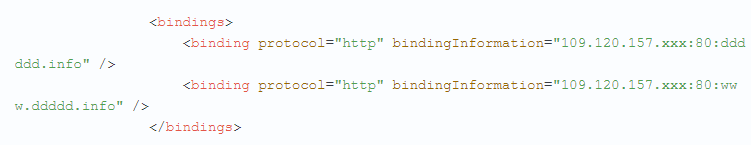
Как отмечалось выше, .NET Framework несёт в себе реализацию компонент для протоколов HTTP/HTTPS (наш любимый ASP.NET), net.tcp, net.pipe и MSMQ. Стеки протоколов HTTP/HTTPS и FTP всё-таки более тесно интегрированы в IIS и ОС, поэтому настройку для нового протокола лучше продемонстрировать на примере менее популярных дотнетовских протоколов. Итак, после установки фреймворка в файле конфигурации IIS ApplicationHost.config появляется записи:



А соответствующие компоненты PPH и ADPH настраиваются в дотнетовском machine.config:



В конфигурационном файле веб-сервера ApplicationHost.config вместе с настройками приложений хранятся связки (bindings), определяющие параметры входящих запросов, которые будут направляться данному приложению. Такими параметрами являются название сетевого протокола, IP-адрес сервера, доменное имя и порт сайта. Эти параметры должны быть уникальными среди работающих приложений для однозначной идентификации целевого приложения. Служба WAS отслеживает это ограничение и не даст вам запустить сайт, у которого это условие не соблюдено, либо предложит остановить сайт с такой же связкой.



Обратите внимание, что в стандартном режиме эксплуатации IIS служба WAS, служба-адаптер для каждого прослушивателя протокола (в т.ч. W3SVC) и сами драйверы/прослушиватели каждого из протоколов (в т.ч. HTTP.sys) запущены в ОС в единственном экземпляре. Но отдельные запросы могут направляться разным приложениям в разных рабочих процессах. С другой стороны, отдельно взятому приложению могут направляться запросы по разным протоколам через соответствующие адаптеры. Видимо, для корректной реализации такого поведения и была придумана архитектурная связка драйвер протокола – адаптер драйвера протокола – служба активации – рабочий процесс.

2.4. Пул приложений

При конфигурации веб-приложения помимо привязок (binding) к параметрам запросов и прочих настроек указывается принадлежность к пулу приложений. Пул приложений стал нововведением в IIS 6 и был призван обеспечить изоляцию веб-приложений друг от друга и тем самым повысить стабильность работы веб-сервера в целом. Суть заключается в том, что код приложения выполняется внутри специального процесса Windows – w3wp.exe. Поэтому исключение внутри веб-приложения приведёт к краху только этого процесса и никак не повлияет на доступность веб-приложений в других пулах и работу служб IIS. Более того, служба WAS попытается заново запустить упавший сайт, и внешние клиенты могут даже не заметить проблем в работе сервера.

Для управления некоторыми параметрами отдельно взятого рабочего процесса w3wp.exe в IIS используется пул приложений. Наиболее часто используемыми из них являются учётная запись, под которой будет запущен процесс, ограничения для очереди запросов, различные таймеры и счетчики для автоматического перезапуска процесса, архитектура x86/x64 (в IIS 7.x) и некоторые другие (рис. 5).

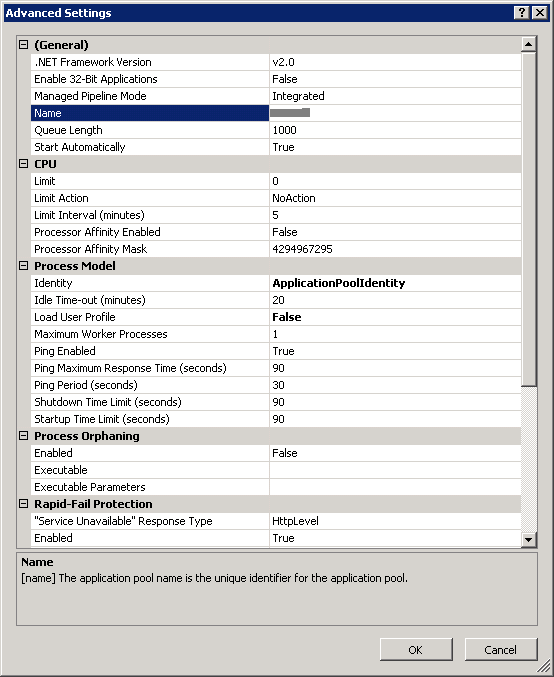


Рис. 5 Дополнительные настройки пула приложений

Ключевым нововведением в концепции пулов приложений в IIS 7.x стал новый параметр – модель управления контейнером, который может принимать 2 значения: классическая (Classic mode) и встраиваемая модель (Integrated mode).

Чтобы объяснить разницу между этими режимами работы, потребуется знакомство с понятием «модуль» (Module) в IIS 6/7.x и событийной моделью обработки запросов в связке IIS + ASP.NET. Тема эта достойна отдельной статьи, но меня на неё уже, увы, не хватит, судя по всему. Здесь же представлю вашему вниманию лишь общие, ключевые моменты.

Итак, IIS при обработке запроса пропускает его внутри рабочего процесса через последовательность специальных компонент – модулей. Например фильтрация, перенаправление, кэширование, аутентификация, авторизация. Каждый такой модуль ассоциируется с определённым событием, а их последовательность составляют событийную модель обработки запросов. Модули делятся на нативные (Native) и управляемые (Managed). Нативные модули поставляются вместе с IIS, а управляемые – в составе .NET Framework (ASP.NET). В общем-то, вы можете управлять ими в определённой степени на уровне конфигурации веб-приложения, но взаимодействовать из кода своего ASP.NET-сайта вы можете только с управляемыми модулями.

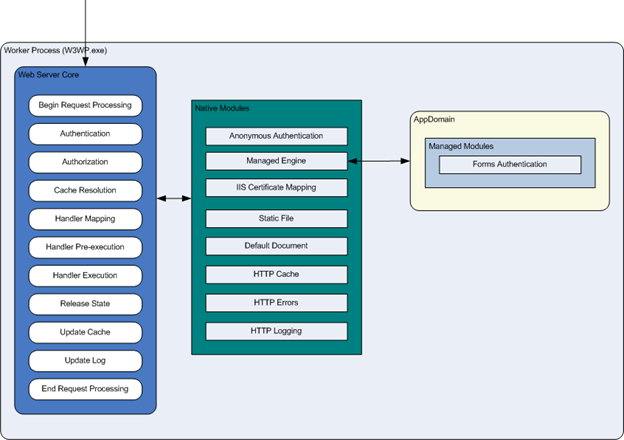


Рис. 6. Идеология модулей в IIS.

Классическая модель управления контейнером обеспечивает обратную совместимость с режимом изоляции рабочих процессов в IIS 6 – запросы к ASP.NET-сайту сначала проходят через нативные модули, а затем передаются в Aspnet\_isapi.dll для обработки модулями в управляемой среде. Такое разделение между IIS и ASP.NET приводит к дублированию некоторых функций, например, аутентификации и авторизации. И вы не имеете возможности управлять программно поведением нативных модулей (пример хоть и не самый животрепещущий, но всё же – раздел «Убираем заголовок Server» в [этой статье](http://habrahabr.ru/post/78213/)).

Встраиваемая модель предполагает более тесное взаимодействие между IIS и ASP.NET. Запрос в такой архитектуре обработки пропускается через установленную последовательность событий, в каждом из которых запрос пропускается через нативный и управляемые модули. В таком режиме модели обработки запросов IIS и ASP.NET объединены в единую модель, что позволяет избежать дублирования функций и получить больший контроль над обработкой запроса.

На практике самое важное, что необходимо учитывать при разработке и развёртывании веб-приложений, – это частичная несовместимость этих двух режимов. Т.е. при переводе сайта (точнее пула приложений, в котором работает сайт) из классической модели во встраиваемую практически всегда потребуется корректировка кода (хоть, возможно, и не значительная), а также тщательное тестирование.

2.5. Домен приложения, приложение

Непосредственными контейнерами веб-приложения являются приложение и домен приложения (Application Domain, AppDomain). Зачастую эти два понятия отождествляются, но всё-таки это немного разные вещи. Приложение – это понятие IIS, а домен приложения – из ASP.NET. Причём в общем случае в приложении может быть несколько доменов. Приложением вы можете управлять из консоли IIS, а доменом приложения – в основном программно. Так, например, перезапускается приложение из консоли. А когда мы пересохраняем web.config, то перезагружается именно домен приложения, не трогая IIS-приложение.

Более важным с практической точки зрения является то, что приложение/домен приложения является sandbox-ом для кода вашего ASP.NET-сайта (не с такой надёжной изоляцией, как в случае с пулом, но всё же).

Ещё один важный момент по умолчанию каждый отдельный рабочий процесс может использовать все имеющиеся на сервере процессоры/ядра, а пул приложений работает на одном рабочем процессе и, следовательно, веб-приложение работает внутри одного IIS-приложения. Но ограничения можно перенастроить.

3. Что дальше?

Дальше нужно разбираться в работе модулей (в терминах IIS) и событийной модели, в которых уже происходит собственно обработка запроса, о чем упоминалось в разделе 2.4. Кратко пройдёмся по основным моментам, которые любопытствующий читатель может проработать самостоятельно.

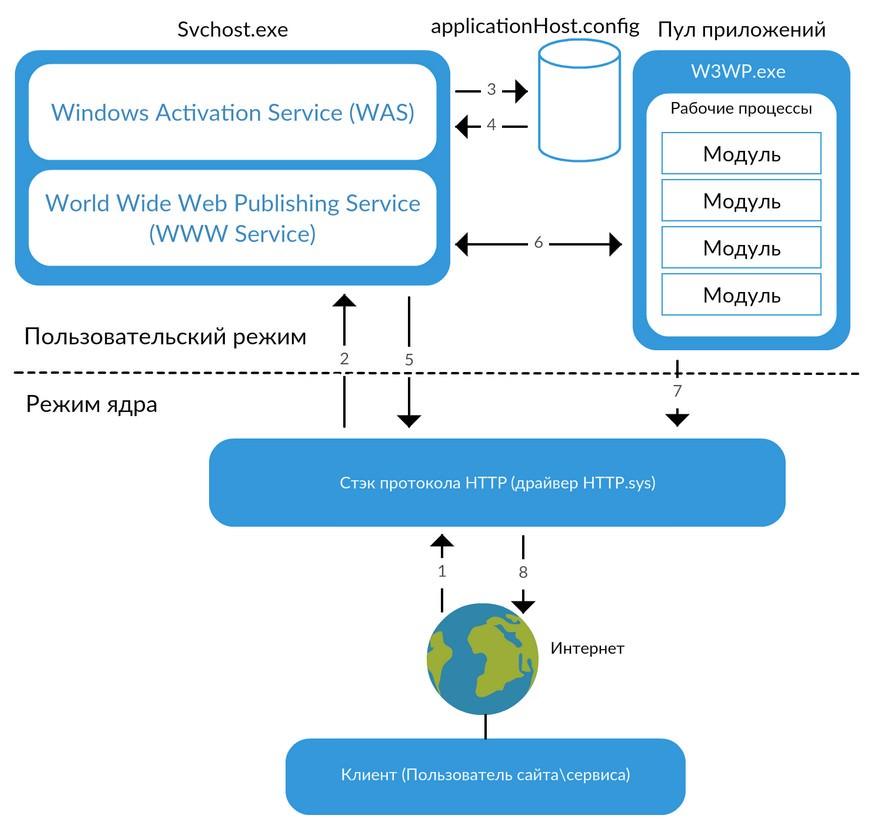
Как отмечалось выше в разделе 2.4 модули IIS содержатся внутри рабочего процесса. Через них последовательно пропускается запрос (в отличие от HttpHandler-ов). Их набор и порядок определяется конфигурацией сервера и/или конкретного веб-приложения. Модули предназначены для отдельных, узконаправленных задач, таких как авторизация, кэширование, кастомное логгирование, сжатие, возврат статического контента и, конечно же, формирование HTML-страниц по заданному URL.

Как мы уже знаем, модули в IIS бывают 2 типов: нативные и управляемые. Точный список модулей вы можете почерпнуть в MSDN-е или в статье Риган Темплин. Вы всегда можете написать свой модуль, например, для редиректов. Чаще всего, конечно, делают управляемые модули, т.к. их проще всего реализовать. К слову, ASP.NET WebForms и MVC работают в виде таких вот управляемых модулей.

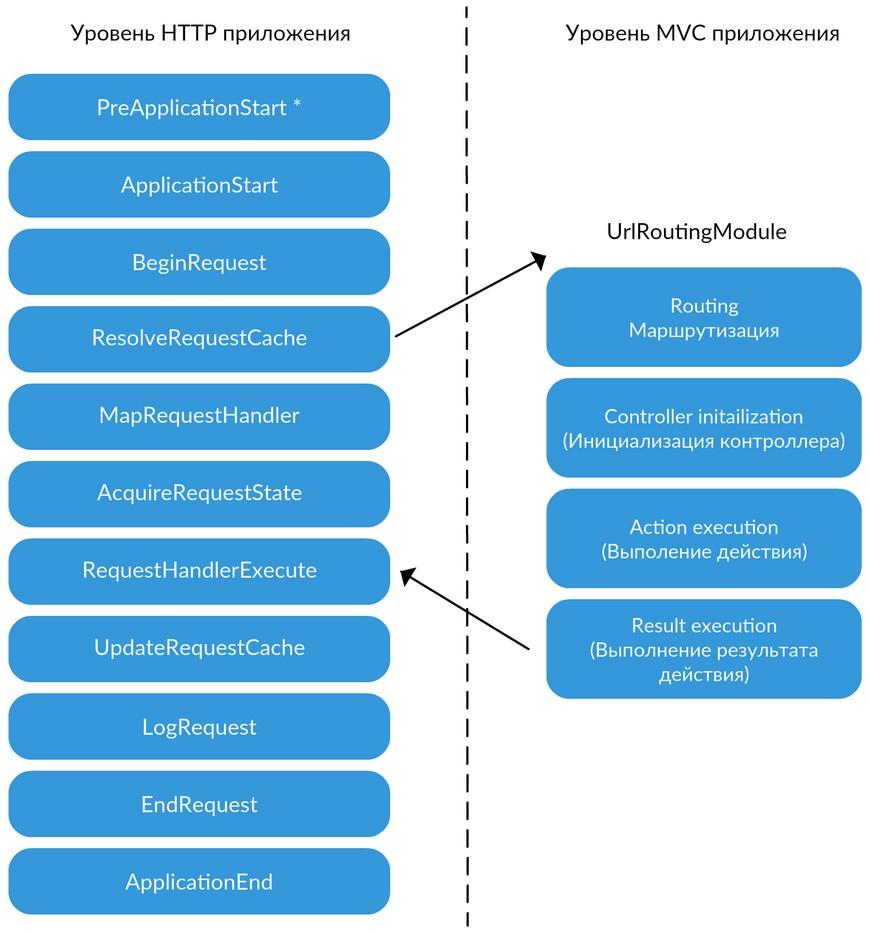
На следующем уровне рассмотрения мы столкнёмся уже с составляющими ASP.NET, такими как HttpHandler-ы и события обработки страницы. Единственное, позволю себе посоветовать идущим на собеседования забить в поисковике «ASP.NET page lifecycle» перед встречей – этого уж точно по моему глубокому убеждению стыдно не знать специалистам, считающих себя разработчиками на ASP.NET.

### 1.9.3 ASP.NET MVC [жизненный цикл запроса](https://d2funlife.com/asp-net-mvc-request-life-cycle)

Основой классического ASP.NET MVC приложения является IIS он и является той самой отправной точкой на длинном пути запроса "от появления на сервере до отправки ответа". Картина прихожей для запросов выглядит следующим образом:



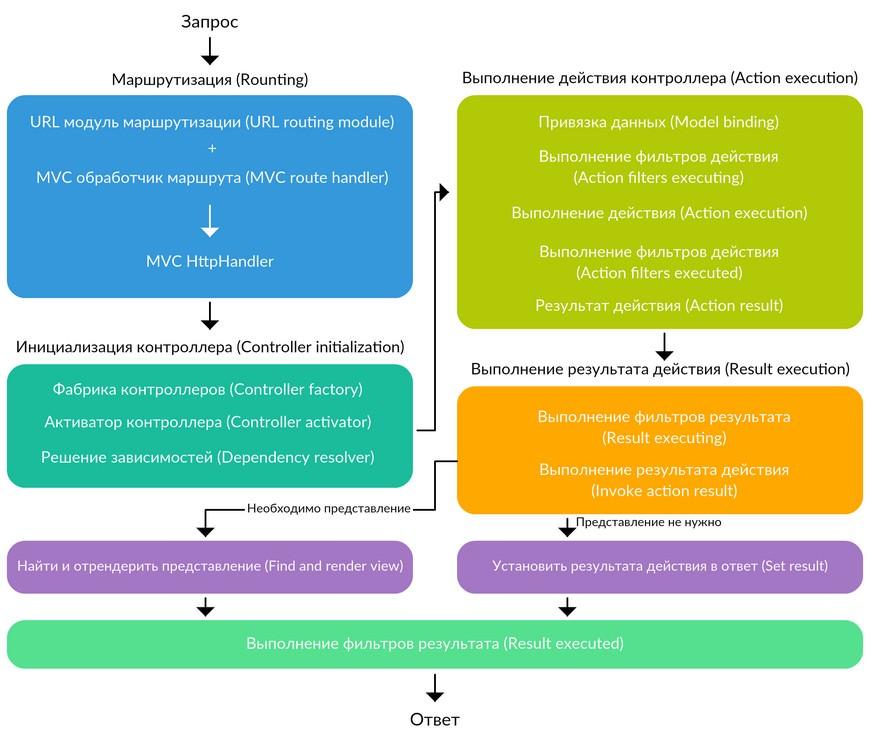
Подробное описание устройства и взаимодействия всех частей цепочки описано ранее. Цель данной статьи рассказать, что именно происходит между 6м и 7м шагом на данной схеме, рассказать про устройство и всю цепочку работы ASP.NET MVC приложения. Путешествие с запросом происходит по следующей цепочке действий:



Здесь видно два "острова", которые соединены мостами. Один из них — это платформа HTTP приложения, если предметнее: открыв Global.asax, мы увидим, что ключевой класс, где происходит конфигурация нашего приложения, наследуется от [HttpApllication](https://goo.gl/WDCEXR) (System.Web). HttpApplication и дает базу для нашего приложения, в том числе и события из списка выше, которые можно использовать из Global.asax.cs, либо подключаться к событиям, реализуя свой HTTP модуль ([IHttpModule](https://goo.gl/xNg7wz)). Пример всех этих точек расширения HttpApplication можно посмотреть тут - [исходный код](https://goo.gl/gkan7r). Используя данный пример, можно в режиме отладки пройти все шаги один за одним и проверить правильность порядка и существование точки расширения в принципе. Уровень HTTP приложения и интеграция его с MVC приложением будет рассмотрен в отдельной статье. В данной статье будет рассмотрен исключительно второй "остров" нашей карты, с его остановками в виде маршрутизации (routing), инициализации контроллера (controller initialization), выполнением действия контроллера (action excecution) и выполнением результата действия (result execution).

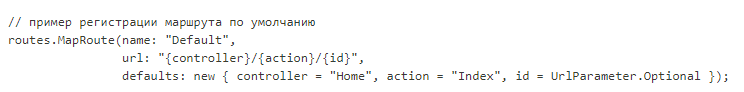
#### 1.9.3.1 MVC цикл

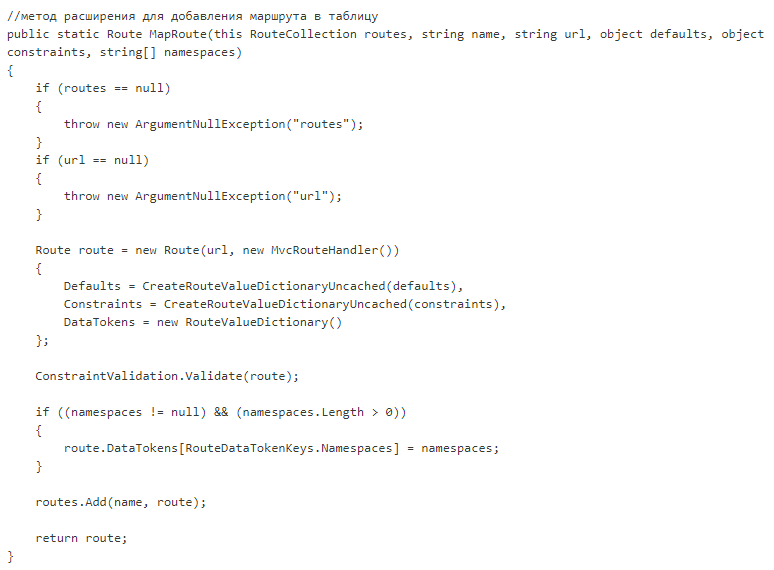
Ранее был представлен общий план цикла запроса. Более детальный цикл уровня MVC приложения выглядит следующим образом:



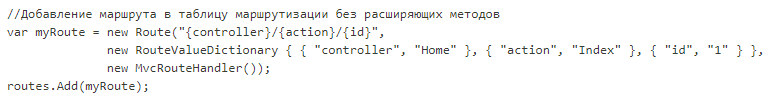
#### 1.9.3.2 Routing (Маршрутизация)

Первой остановкой запроса является подсистема маршрутизации, ведь необходимо определить, какой цели должен достигнуть наш запрос, какое действие и какого контроллера является для него желанным, а может быть запросу вовсе нужен статический файл, а не контроллер, в любом случае необходимо получить некую пару "ключ – значение" для маршрута и единицы, которая будет его обрабатывать дальше. Эта единица называется "обработчик запроса" (route handler) и ее задача вернуть объект [HttpHandler](https://goo.gl/DVGt6e), который будет выполнять всю дальнейшую цепочку обработки запроса. Неопытный программист задастся вопросом "какой route handler? Ведь мы при старте приложения просто указываем общий шаблон в текстовом варианте и все. Там нет указания каких-либо route handler, только имя контроллера и действия по умолчанию". И здесь тот момент, когда оптимизации и улучшения сыграли злую шутку с программистом и он не знает внутренних тонкостей из-за того, что их скрыли от него. Пример объявления маршрутизации по умолчанию:

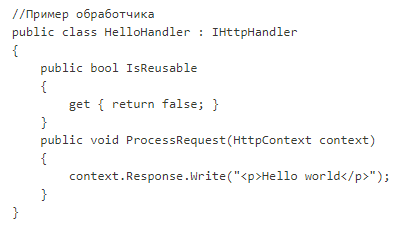


Все красиво, все просто: передаем имя маршрута, шаблон url запроса, анонимный объект, который указывает названия контроллера и действия по умолчанию. Но MapRoute – это расширяющий метод, который скрывает за собой реальную логику добавления маршрута в таблицу маршрутизации.

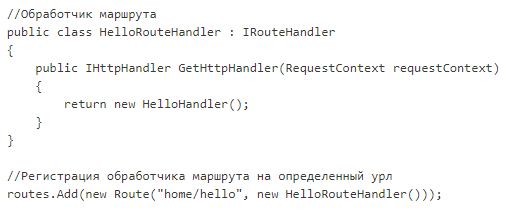
Пример добавления маршрута в таблицу маршрутизации без использования расширяющих методов:



Вот тут уже более наглядно будет видно, что за route handler и как они вообще попадают в маршрутизацию. Сам класс обработчика маршрута реализует интерфейс [IRouteHandler](https://goo.gl/FFy1yZ) у которого есть единственный метод GetHttpHandler, возвращающий HttpHandler. HttpHandler – это тот объект который берет на себя ответственность за генерацию ответа на HTTP запрос, класс этого объекта реализует интерфейс IHttpHandler. Этот интерфейс содержит свойство IsReusable и метод ProcessRequest. IsReusable - флаг, указывающий может ли другой запрос использовать этот же обработчик, иными словами нужно ли ему создавать каждый раз новый экземпляр или можно создать один раз и использовать его для всех последующих запросов. ProcessRequest – метод, который выполняет обработку запроса. По умолчанию используется [MvcRouteHandler](https://goo.gl/ZxW4Wi) и [MvcHandler](https://goo.gl/XRwb9k). Исходный код открывает все тонкости и оптимизации, которые заложили архитекторы. В реальной жизни очень мала вероятность , что придется создавать собственный HttpHandler, так как стандартные средства ASP.NET MVC покрывают большинство задач. Приведу пример маленького HttpHandler`а, который поздоровается с вами:



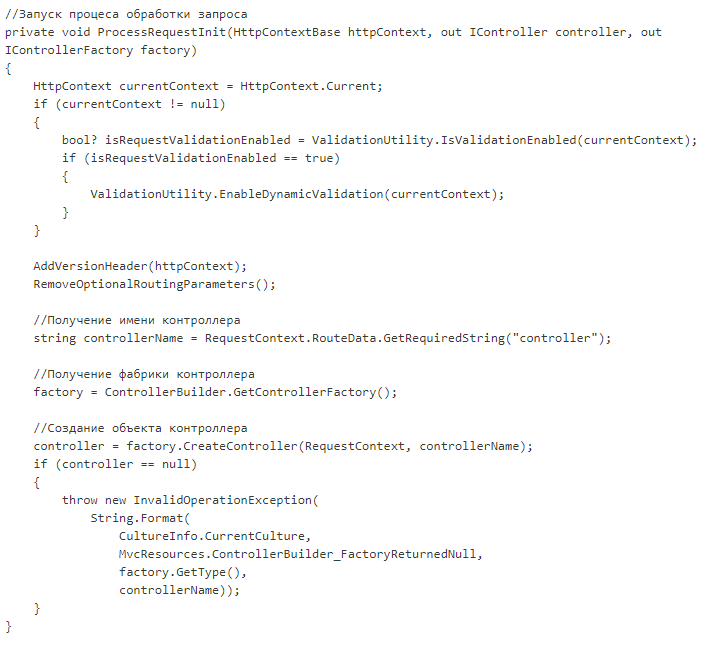
Получить данный обработчик мы можем только из RouteHandler, поэтому сделаем его и зарегистрируем по адресу "home/hello".



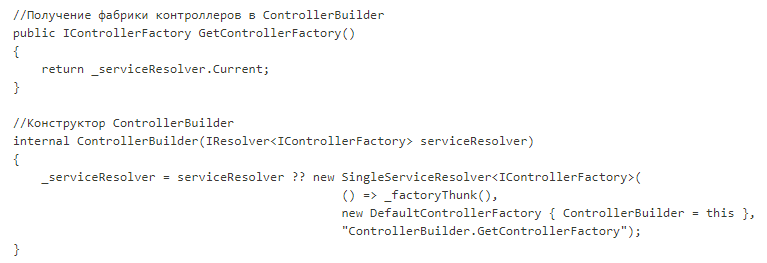
Таким образом мы получим приветствие, которого ждали, но это лирическое отступление с демонстрацией точки расширения в ASP.NET MVC, где вы можете управлять процессом обработки запроса. В стандартном приложении мы получаем [MvcRouteHandler](https://goo.gl/yF9rog), который отдает [MvcHandler](https://goo.gl/5uK7Cf), а тот в свою очередь запускает нужный контроллер, нужное действие и отдает им нужные данные.

#### 1.9.3.3 Controller initialization (Инициализация контроллера)

Как было уже сказано ранее, по умолчанию мы получаем MvcRouteHandler, он в свою очередь отдает MvcHandler. MvcHandler в методе ProcessRequest начинает процесс создания нужного контроллера и исполнение нужного действия.

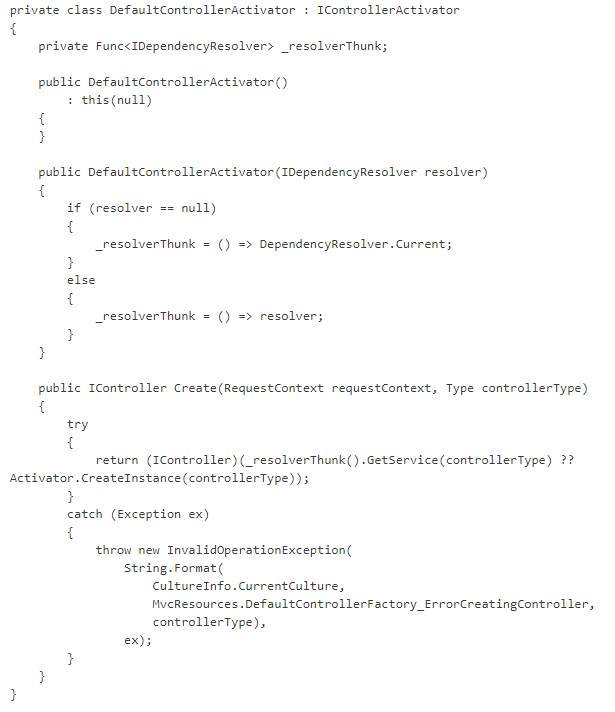


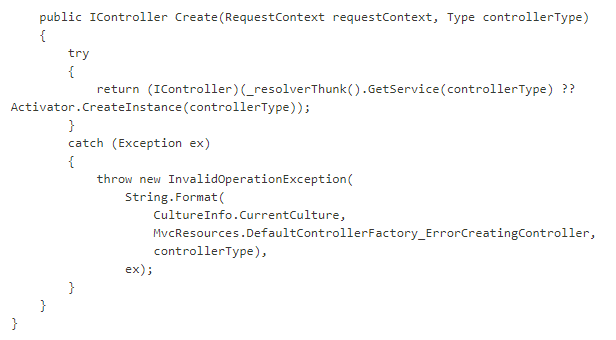
В MVCHandler мы получаем объект фабрики контроллеров (controller factory), который ответственен за получение контроллера и реализует интерфейс [IControllerFactory](https://goo.gl/icTch6). По умолчанию [ControllerBuilder](https://goo.gl/3VQZLX) отдает объект класса [DefaultControllerFactory](https://goo.gl/y1e4LK). Увидеть это мы можем, посмотрев исходный код:

В методе CreateController используется объект [IControllerActivator](https://goo.gl/ysU8RF) в методе GetControllerInstance, такое разграничение дополнительным уровнем дает возможность использовать разрешение зависимостей при создании контроллеров.

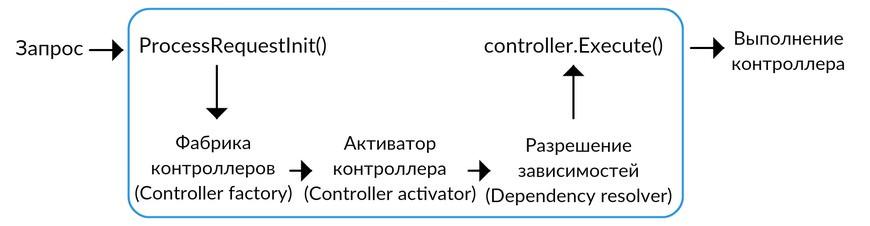


Активатор контроллера по умолчанию реализован в классе [DefaultControllerActivator](https://goo.gl/C1Pb8G) и реализация получения объекта с использованием разрешения зависимостей (dependency resolver) выглядит следующим образом:





\_resolveThunk по умолчанию получает объект класса [DefaultDependencyResolver](https://goo.gl/zkRKxM), который является оберткой для System.Activator, но если вы подключите собственный dependency resolver, то использоваться будет именно он. Используя различные контейнеры инверсии зависимостей ([IoC containers](https://goo.gl/h8X1PQ)) + внедрение зависимостей (DI), мы расширяем наши контроллеры и можем дополнительно управлять жизненным циклом объектов. Интеграция контейнера присходит посредством класса [DependencyResolver](https://goo.gl/8FuWbQ), он статичен и имеет метод SetResolver, где и будет установлен наш IoC контейнер, который, в свою очередь, будет получен в активаторе контроллера в конструкторе (тот самый \_resolveThunk). И вот, когда контроллер получен, MvcHandler может вызывать метод Execute у контроллера, передав объект [RequestContext](https://goo.gl/f7XE6y) (контекст запроса) в качестве аргумента. Схематически, в сжатом виде, инициализация контроллера выглядит следующим образом:



#### 1.9.3.4 Action execution (Выполнение действия)

При выполнении действий контроллера, так же выполняются фильтры. Фильтры — это точки расширения, которые позволяют обогатить код дополнительной логикой. Существуют следующие типы фильтров authentication filters (фильтры аутентификации), authorization filters (фильтры авторизации), action filters (фильтры действий), result filters (фильтры результата действия), exception filters (фильтры исключений). Фильтры действий реализуют интерфейс [IActionFilter](https://goo.gl/g81jLh), в котором определены два метода OnActionExecuting и OnActionExecuted, соответственно они выполняются перед выполнением действия и после выполнения действия. При выполнении цикла запроса фильтры исполняются в следующем порядке:

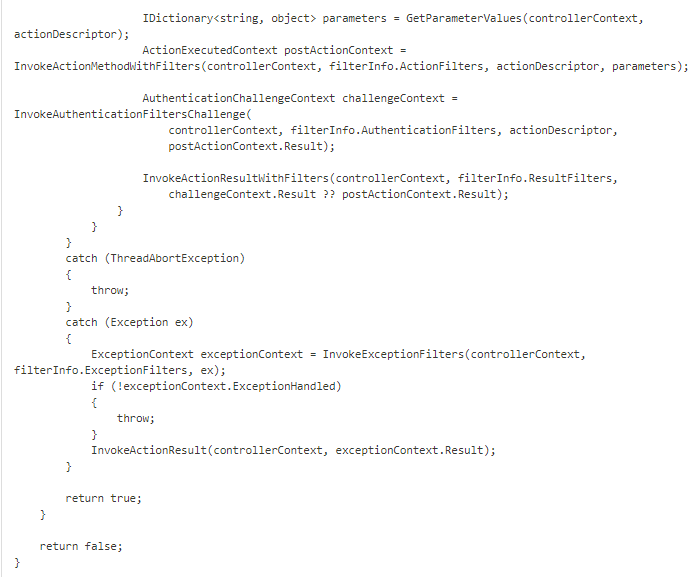
1. Фильтры аутентификации
2. Фильтры авторизации
3. Фильтры действий
4. Фильтры результата действия

Фильтры исключений выполняются на любом этапе, ведь ошибки нужно отлавливать и обрабатывать по всему приложению.

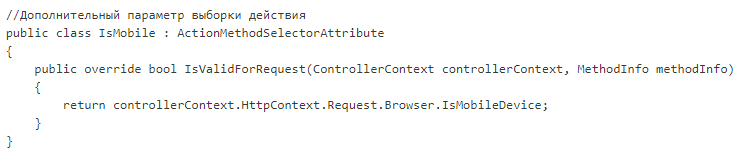
В исходном коде класса Controller мы видим, что [Controller](https://goo.gl/pZ1Fsy) наследует [ControllerBase](https://goo.gl/kvkWUW), в ControllerBase реализован метод Execute и в нем уже вызывается ExecuteCore. ExecuteCore в свою очередь реализован в классе Controller, получив имя действия для выполнения, запускает его на выполнение с исполнением дополнительного уровня в виде [ActionInvoker](https://goo.gl/71nzQn). ActionInvocker в себе инкапсулирует выполнение дополнительных действий (привязку данных, выполнение всех фильтров, запуск на исполнение результата действия) и выполнение самого действия. Все происходит в следующем порядке внутри метода InvokeAction:

1. Фильтры аутентификации (метод InvokeAuthenticationFilters)
2. Фильтры авторизации (метод InvokeAuthorizationFilters)
3. Получение значений параметров, если формальнее "привязка модели" (метод GetParameterValues)
4. Выполнение действия и фильтров действия (метод InvokeActionMethodWithFilters)
5. Выполнение результата действия (метод InvokeActionResultWithFilters)





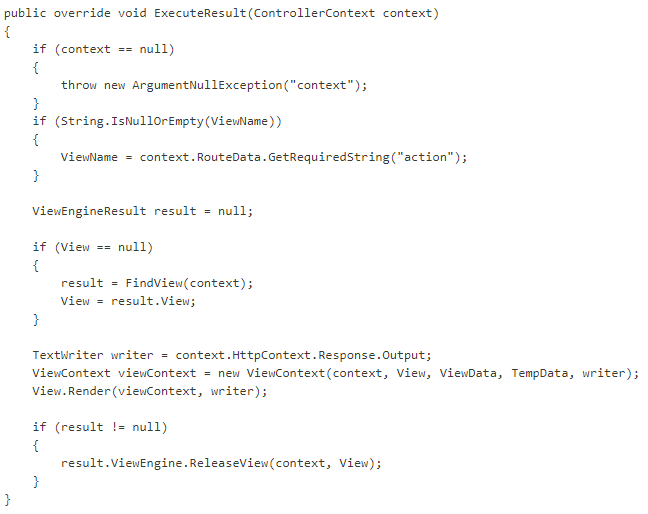
ActionInoker определяет, какое действие необходимо запустить, для этого берутся за ключевые фильтры: имя действия, параметры действия, атрибуты [ActionMethodSelectorAtribute](https://goo.gl/fVVW4f) (если добавлены). Такими атрибутами являются [HttpGet](https://goo.gl/BUKEmL), [HttpPost](https://goo.gl/p9Vfjj), к примеру, вы можете добавить свой, для этого просто необходимо создать класс, который наследует ActionMethodSelectorAtribute и переопределяет метод IsValidForRequest. Пример атрибута, который уточнит выбор действия, определив является ли клиент мобильным браузером или нет:



Если при выборке необходимого действия было найдено более одного или ни одного, то будет ошибка выполнения, ведь явное одно действие не было выбрано. Отдельно хочу отметить, что этап привязки данных к модели описан в статье ASP.NET MVC привязка данных, тут можно подчерпнуть информацию и посмотреть примеры, как мы можем влиять на привязку модели.

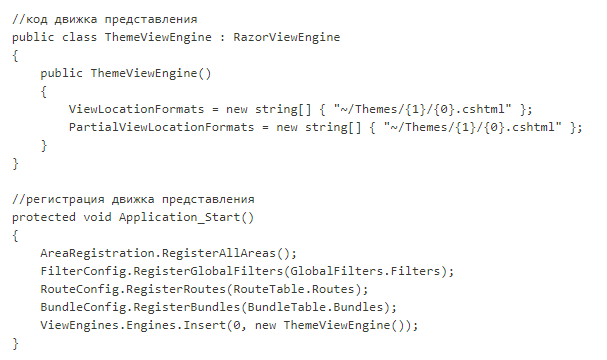
#### 1.9.3.5 Result execution (Выполнение результата действия)

В результате выполнения всей цепочки мы получаем объект ActionResult, который вместе с фильтрами результата выполняется в методе InvokeActionResultWithFilters. Данный метод вызывает InvokeActionResultFilterRecursive в котором рекурсивно выполняются фильтры результата и выполняется сам результат. Что значит выполняется сам результат? Объект [ActionResult](https://goo.gl/JN5tvf) записывает свои данные в ответ запросу. Если действие возвращает JsonResult, то в ответ будет записан сериализованный в json объект, который отдан в результат действия. Если же результатом нашего действия является представление или частичное представление, то будет выполнена следующая цепочка действий:

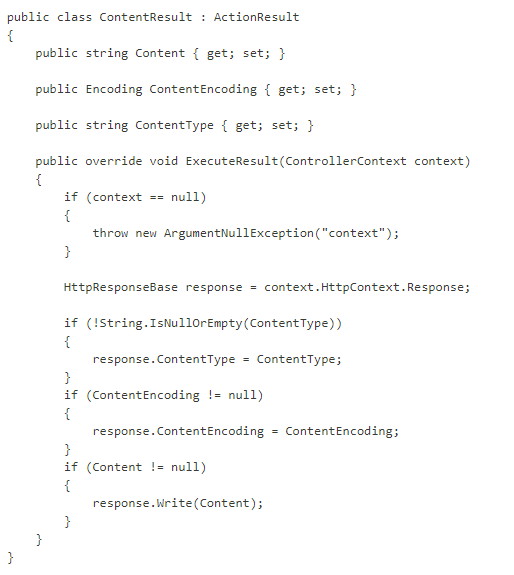
Как видно, тут происходит следующее:

1. В базовом классе ViewResultBase вызывается метод ExecuteResult
2. Происходит поиск представления для рендера (реализован в классе [ViewResult](https://goo.gl/NxkAJ5)) по всем движкам представлений (ViewEngines)
3. Найденное представление отправляется на рендер
4. Результат рендера записывается в ответ на запрос

Есть в этой цепочке один нюанс, мы можем определить свой ViewEngine с путями поиска представлений и добавить его к списку стандартных. Таким образом мы сможем расширить правила поиска представлений. Ключевой момент: при поиске представления, проходит цикл по всем зарегистрированным ViewEngines и первый, который сможет найти представление возвращает результат, то есть для указания более частного случая необходимо свой ViewEngine регистрировать в списке на более ранних позициях.



Данный пример объясняет цепочку вызовов, когда результатом является представление, если у нас более простой ответ, к примеру, ContentResult, то вся цепочка выглядит куда проще:



Вот так мы и подошли к концу цикла запроса в MVC, в следующей статье завершим детальный обзор цикла HTTP приложения, получив полную картину, как работает само приложение, как обрабатываются запросы от пользователей. Если вы захотите сами побродить по просторам MVC приложения, то это можно сделать, скачав код тут : [https://aspnetwebstack.codeplex.com](https://aspnetwebstack.codeplex.com/) или тут : <https://github.com/ASP-NET-MVC/aspnetwebstack>. Сodeplex уходит в прощальную гастроль, но официальный репозиторий пока только там, на github не завезли еще, только не официальное зеркало. Также посмотрите подробные диаграммы жизненного цикла от Microsoft [Lifecycle of an ASP.NET MVC 5 Application](https://goo.gl/o7Rd2b)

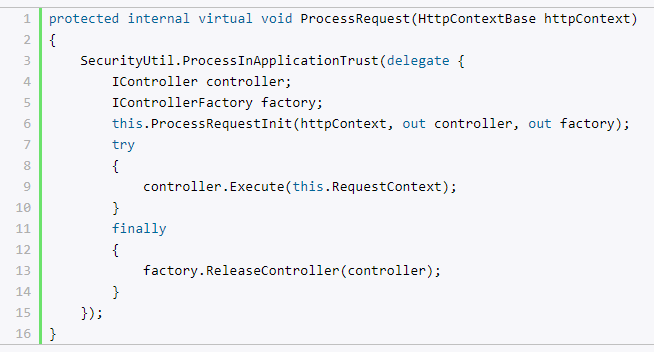
### 1.9.4 Жизненный цикл [приложения ASP.NET MVC](https://metanit.com/sharp/articles/mvc/13.php)

Отправляя из браузера запрос к приложению, нередко мы практически моментально получаем нужный контент. Однако в реальности обработка запроса проходит кучу различных этапов. И в этой статье мы рассмотрим основные этапы жизненного цикла приложения на ASP.NET MVC 5.

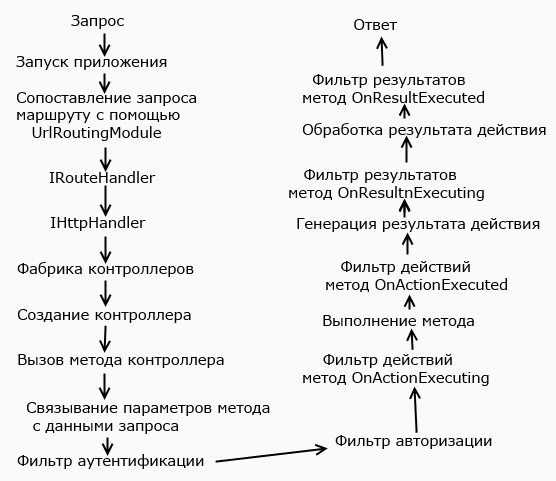
1. После получения IIS запроса на обработку, при первом обращении к ресурсу, работающему под управлением CLR, создается объект класса ApplicationManager, который представляет домен приложения, внутри которого обрабатывается запрос. Домены приложений изолируют выполняющиеся приложения друг от друга. Уже внутри домена приложения создается объект класса HostingEnvironment, который предоставляет доступ к информации о приложении, в частности, он сообщает имя и каталог приложения.
2. После создания домена приложения также создаются и инициализируются такие объекты, как HttpContext, HttpRequest и HttpResponse, которые инкапсулируют всю информацию, связанную с текущим запросом к приложению.
3. На следующей стадии уже непосредственно запускается приложение, которое представляет экземпляр класса HttpApplication. Если в приложении определен файл Global.asax, то среда ASP.NET в качестве приложения создает объект класса Global.asax, который в свою очередь наследуется от класса HttpApplication. На этой же стадии происходит начальная инициализация приложения в методе Application\_Start, который находится в файле Global.asax.cs.
4. После этого запрос начинает обрабатываться в конвейере класса HttpApplication - начинается собственно обработка запроса приложением. И первым шагом здесь является установка маршрута. Запрос перехватывается специальным HTTP-модулем под названием UrlRoutingModule. Этот модуль выбирает маршрут, который соответствует входящему запросу.  
   Весь набор маршрутов определяется в файле RouteConfig.cs:

  
А сама регистрация маршрутов происходит в методе Application\_Start строкой RouteConfig.RegisterRoutes(RouteTable.Routes);. Для каждого приложения имеется только один объект коллекции маршрутов RouteTable.

1. Если модулю UrlRoutingModule удастся сопоставить запрос с одним из маршрутов в коллекции RouteTable, то затем UrlRoutingModule выбирает обработчик маршрутов сопоставленного маршрута - объект IRouteHandler. По умолчанию объект IRouteHandler представляет экземпляр класса MvcRouteHandler.
2. Затем у объекта IRouteHandler вызывается метод GetHandler, который возвращает объект интерфейса IHttpHandler, используемый для обработки запроса. По умолчанию в качестве IHttpHandlera используется объект класса MvcHandler.
3. У обработчика IHttpHandler вызывается метод ProcessRequest для обработки запросов:

  
На этом этапе уже происходит непосредственно создание контроллера, который представляет объект интерфейса IController. За создание контроллера отвечает объект интерфейса IControllerFactory, представляющий фабрику контроллеров. В качестве класса по умолчанию для фабрики контроллеров выступает класс System.Web.Mvc.DefaultControllerFactory

1. После этого начинается собственно выполнение кода контроллера. После инициализации для запуска метода для обработки запроса контроллер вызывает метод InvokeAction():
2. Далее привязчик модели (по умолчанию класс System.Web.Mvc.DefaultModelBinder) извлекает данные из запроса, производит из преобразование, форматирование, валидацию и связывает их с определенными параметрами вызываемого метода.
3. При вызове метода в ASP.NET MVC 5 запускается фильтр аутентификации, представляющий объект интерфейса IAuthenticationFilter. Он аутентифицирует пользователя
4. После фильтра аутентификации в ASP.NET MVC 5 запускается фильтр авторизации, представляющий реализацию интерфейса IAuthorizationFilter. До MVC 5 фильтры аутентификации и авторизации объединялись в один фильтр и срабатывали вместе. Фильтр авторизации управляет доступом пользователя к определенным ресурсам на основе его учетных данных.
5. Перед непосредственным выполнением метода контроллера запускается метод OnActionExecuting фильтра действий. Фильтр действий представляет объект интерфейса IActionFilter. Кроме того, также после выполнения метода контроллера запускается другой метод фильтра действий - метод OnActionExecuted
6. Собственно выполнение метода контроллера. Он выполняет определенную логику и на выходе генерирует результат обработки в виде объекта ActionResult.
7. При обработке результата срабатывает другой фильтр - фильтр результатов - объект интерфейса IResultFilter. Его метод OnResultExecuting срабатывает до обработки результата, а метод OnResultExecuted после.
8. Генерация результата представляет создание объекта одного из классов результатов действий - ViewResult, ContentResult, FileResult, RedirectResult и др.  
   На заключительном этапе у каждого объекта ActionResult вызывается метод ExecuteResult, который обрабатывает результат действия. Для объектов ViewResult и PartialViewResult это выражается в поиске необходимого представления и его рендеринге движком представлений (как правило, движком Razor), который представляет объект интерфейса IViewEngine. Результат обработки в виде html-страницы посылается пользователю.  
   Для других объектов ActionResult (ContentResult, RedirectRefult и др.) происходит простая отправка результата в выходной поток.
9. И в конце пользователь получает результат обработки своего запроса.

Схематично весь процесс конвейера приложения на ASP.NET MVC 5 можно представить так:

### 1.9.5 Дополнительная инфа

#### 1.9.5.1 [ASP.NET Advanced. Урок 1. Архитектура IIS. Жизненный цикл HttpApplication.](https://www.youtube.com/watch?v=qiujcPFFinA)

#### 1.9.5.2 Н[абором статей по этой теме](http://www.msdr.ru/articles.aspx)

# 3. Создание MVC приложения

## 3.1 Шаблоны MVC приложения(Project Template)

Empty - cоздает пустой проект, в котором нету ничего кроме заранее предусмотренных директорий;

Basic - кроме директорий добавляются библиотеки, например, JQuery для валидации пользовательских данных и тд;

Internet Application

Single Page Application - используется Ajax для создания одностроничного сайта

## 3.2 Структура проекта MVC 5 MVC в (Solution explorer)

App\_Data: содержит файлы, ресурсы и базы данных, которые используются приложением;

App\_Start: хранит ряд статических файлов, которые содержат логику инициализации приложения при запуске:

- BundleConfig.cs: файл, в котором настраиваются бандлы, которые помогают организовать файлы скриптов и стилей более эффективным путем для снижения издержек при передаче клиенту;

- FilterConfig.cs: файл,в котором настраиваются глобальные фильтры;

- RouteConfig.cs: файл, в котором настраиваются маршруты.

Content: содержит вспомогательные файлы, которые не включают код на c# или javascript, и которые развертываются вместе с приложением, например, файлы стилей css;

Controllers: содержит файлы классов контроллеров. По умолчанию в эту папку добавляются два контроллера - HomeController и AccountController(ток почему-то у меня не создался AccountController);

fonts: хранит дополнительные файлы шрифтов, используемых приложением;

Models: содержит файлы моделей. По умолчанию Visual Studio добавляет пару моделей, описывающих учетную запись и служащих для аутентификации пользователя;

Scripts: каталог со скриптами и библиотеками на языке javascript;

Views: здесь хранятся представления. Все представления группируются по папкам, каждая из которых соответствует одному контроллеру. После обработки запроса контроллер отправляет одно из этих представлений клиенту.

Views/Shared: в ней находятся представления общие для всех контроллеров, например:

\_Layout.cshtml - это шаблон в который внедряется разметка каждой страницы которая есть в нашем приложении т.е глобальная разметка всех страниц.

Error.cshtml - используется если с в приложении где-то произошла ошибка то будет использоваться этот файл

Global.asax: файл, запускающийся при старте приложения и выполняющий начальную инициализацию. Как правило, здесь срабатывают методы классов, определенных в папке App\_Start. Представляет в целом HTTP приложение т.е. когда сайт запускается на стороне сервера сайт представляется экземпляром класса HTTP Application, который наследуется от HttpApplication. Файл Global.asax необходим чтобы обрабатывать жизненный цикл приложения.

packages.config: файл, который содержит установленные в проект пакеты Nuget

Web.config: файл конфигурации приложения

## 3.3 Соглашения именования

### 3.3.1 Соглашение первого вида

это те соглашения, которые можно нарушать, например, именование папки Script в этой папке должны быть расположены JS сценарии. Нарушить это можно, например, создав папку MyScript и поместить туда JS сценарии. Тоже самое и для папки Content, в которой расположены изображения, Css файлы и т.д.

### 3.3.2 Соглашение второго вида

Эте те соглашения которые нельзя нарушать.

Классы контроллера должны иметь имена, заканчивающиеся словом Controller. Например HomeController, MyController.

Представления должны располагаться в папке /Views/Name\_Controller. Например View для HomeController должно находиться в папке /Views/Home/. имя View может быть каким угодно, но по умолчанию совпадает с именем действия.

# 

# 4. Разработка моделей Asp.NET MVC

## 4.1 Понятие модели

Модель содержит всю логику, относящуюся к логики валидации и доступа к данным. Например, если вы используете Microsoft Entity Framework для доступа к базе данных, вы можете создать классы EF в папке Models. Итак, представление должно содержать логику пользовательского интерфейса, контроллер – обработку, все остальное должно содержаться в модели.

В целом, вы должны стремиться к стратегии «толстых» моделей и «тонких» контроллеров. Методы контроллеров должны содержать несколько линий кода. Если метод контроллера становится слишком «толстым», нужно подумать о переносе логики в новый класс в папке Models.

## 4.2 Создание модели (нет ответа)

• Проектирование модели  
• Использование аннотаций данных на свойствах модели.  
• Валидация пользовательского ввода с помощью аннотаций данных.  
• Расширение моделей. Понятие сервисов.  
• Демонстрация практической реализации модели и ее использование

# 5. Разработка контроллеров Asp.NET MVC

Контроллеры ответственны за обработку пользовательского взаимодействия и содержат бизнес-логику приложения, отвечая за то, что будет выдано пользователю при его запросе. Контроллер - это обычный класс, который наследуется от системного класса Controller и в котором находятся открытые методы. Все открытые методы контроллера называются методами действия. Эти методы можно запустить если выполним обращение к нашему приложению т.е сделаем HTTP запрос.

Фреймворк MVC связывается URL-ы с классами-контроллерами. Контроллеры обрабатывают входящие запросы, пользовательский ввод, выполняют логику приложения. Класс контроллера обычно генерирует отдельный компонент представления.

Базовым классом ко всем контроллерам является ControllerBase, предоставляющий базовую функциональность обработки. Класс Controller наследуется от ControllerBase и является стандартной реализацией контроллера, отвечает за:

* Определение подходящего метода для вызова и валидация возможности его вызова
* Получение значений для использования их в качестве аргументов методов
* Обработка ошибок выполнения

## 5.1 Взаимодействие контроллера с пользовательским запросом.

В приложениях ASP.NET, не использующих фреймворк MVC (ASP.NET Routing), взаимодействие с пользователем организовано с помощью страниц и событийной модели. Вместо этого, в приложениях ASP.NET MVC используются контроллеры и методы. Контроллер определяет методы. Контроллеры могут иметь много методов.

Методы обычно связываются с пользовательским действием связью один-к-одному. Пример пользовательского действия: пользователь вводит URL в браузер, нажимает ссылку и отправляет форму. Каждое из этих действий отправляет запрос на сервер. В каждом случае URL запроса содержит информацию, которую фреймворк использует для вызова метода.

## 5.2 Маршрутизация запроса

Когда вводится URL, приложение MVC использует правила маршрутизации, определенные в файле Global.asax, для обработки URL и определения пути контроллера т.е запрос браузера связывается с методом контроллера с помощью функциональности фреймворка под названием ASP.NET Routing. Эта функциональность используется фреймворком для маршрутизации входящих запросов и связывания их с соответствующими методами контроллеров. Маршрутизация использует таблицу маршрутизации, которая содержится в файле Global.asax.

При запуске приложения вызывается метод Application\_Start(). В листинге 1 этот метод вызывает метод RegisterROutes(), создающий стандартную таблицу маршрутизации. Стандартная таблица маршрутизации состоит из одного маршрута, и этот маршрут разбивает все входящие запросы на три сегмента. Первый сегмент связывается с именем контроллера, второй – с его методом, и третий – с параметром, который передается в метод.

Например, URL:

/Product/Details/3

Будет разбит на:

Контроллер = Product

Метод = Details

Параметр Id = 3

Стандартный маршрут определен в файле Global.asax для всех трех параметров. Стандартный контроллер – Home, метод – Index, ID по умолчаию пустой.

Наконец, если вы запустите приложение ASP.NET MVC без поддерживаемых типов URL(<http://localhost/>), URL будет разбит на:

Контроллер = Home

Метод = Index

Параметр Id = пусто

Запрос будет связан с методом Index() контроллера HomeController.

### 5.2.1 Определение действий контроллера.

После этого контроллер определяет подходящий метод для обработки запроса. По умолчанию, URL запроса воспринимается как субпуть, включающий имя контроллера и имя метода. Например, если пользователь ввел URL http://contoso.com/MyWebSite/Products/Categories, субпуть будет /Products/Categories. Стандартное правило маршрутизации воспринимает Products как префикс имени контроллера, которое должно заканчиваться “Controller” (ProductsController). “Categories” воспринимается как имя метода. Поэтому правило маршрутизации вызывает метод Categories контроллера Products. Если URL выглядит как /Products/Detail/5, то в метод Detail передается в качестве параметра 5.

### 5.2.2 Виды возвращаемого результата.

Большинство методов возвращаются экземпляр класса, наследуемого от ActionResult, класса, являющегося базовым для всех результатов методов. Однако существуют другие типы результатов, которые можно использовать в зависимости от задачи метода. Например, самый популярный метод – вызов метода View. Метод View возвращает экземпляр класса ViewResult, наследуемый от ActionResult.

Вы можете создавать методы, возвращаемые объект любого типа (строку, число, булево). Эти типы будут обернуты в подходящий тип ActionResult перед тем, как будет сгенерированы в ответное действие.

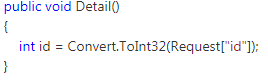
|  |  |  |
| --- | --- | --- |
| Action Result | Метод | Описание |
| [ViewResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.viewresult(v=VS.98).aspx) | [View](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.view(v=VS.98).aspx) | Генерирует представление как веб-страницу. |
| [PartialViewResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.partialviewresult(v=VS.98).aspx) | [PartialView](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.partialview(v=VS.98).aspx) | Генерирует частичное представление, определяемое часть представления, которая будет сгенерирована внутри другого представления. |
| [RedirectResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.redirectresult(v=VS.98).aspx) | [Redirect](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirect(v=VS.98).aspx) | Перенаправление на другой метод с указанием его URL. |
| [RedirectToRouteResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.redirecttorouteresult(v=VS.98).aspx) | [RedirectToAction](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirecttoaction(v=VS.98).aspx)  [RedirectToRoute](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirecttoroute(v=VS.98).aspx) | Перенаправление на другой метод. |
| [ContentResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.contentresult(v=VS.98).aspx) | [Content](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.content(v=VS.98).aspx) | Возвращение определенного пользователем типа контента. |
| [JsonResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.jsonresult(v=VS.98).aspx) | [Json](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.json(v=VS.98).aspx) | Возвращение сериализованного объекта JSON. |
| [JavaScriptResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.javascriptresult(v=VS.98).aspx) | [JavaScript](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.javascript(v=VS.98).aspx) | Возвращение скрипта, выполняемого на стороне клиента. |
| [FileResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.fileresult(v=VS.98).aspx) | [File](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.file(v=VS.98).aspx) | Возвращение бинарного потока. |
| [EmptyResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.emptyresult(v=VS.98).aspx) | (None) | Возвращение результата в том случае, если метод возвращает null (void). |

### 5.2.3 Использование параметров в действиях

По умолчанию значения для параметров методов получаются из коллекции данных запроса. Колекция данных содержит пары значений имя-значения. Класс контроллера определяет метод и значения в качестве его параметров, основываясь на экземпляре RouteData и данных из запроса. Если значение параметра не может быть обработано и тип параметра является ссылочным или nullable типом, в качестве значения параметра передается null. Иначе выбрасывается исключение.

Есть несколько путей доступа к параметрам в методах классов контроллеров. Класс Controller предоставляет [Request](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.request(v=VS.98).aspx) и [Response](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.response(v=VS.98).aspx), которые имеют такой же смысл, как [HttpRequest](http://msdn.microsoft.com/en-us/library/system.web.httprequest(v=VS.98).aspx) и [HttpResponse](http://msdn.microsoft.com/en-us/library/system.web.httpresponse(v=VS.98).aspx) , части ASP.NET. Однако эти объекты принимают объекты, реализующие абстрактные классы [HttpRequestBase](http://msdn.microsoft.com/en-us/library/system.web.httprequestbase(v=VS.98).aspx) и [HttpResponseBase](http://msdn.microsoft.com/en-us/library/system.web.httpresponsebase(v=VS.98).aspx). Эти базовые классы предоставляют возможность создания объектов-«болванок», что приводит к возможности эффективного и легкого тестирования классов контроллеров.

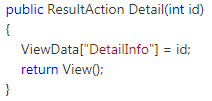
Пример того, как получить с помощью объекта Request строкове значение id:



#### 4.2.3.1 Автоматическое связывание параметров методов

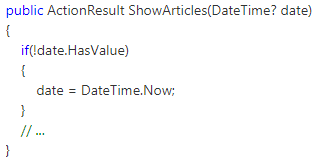
Фреймворк MVC автоматически связывает значения параметров URL с параметрами методов и, по умолчанию, если метод принимает параметр, фреймворк осматривает входящие данные запроса и определяет, содержит ли запрос значение с подобным именем и, если да, то связывает.

Следующий пример показывает один из вариантов предыдущего примера, в котором параметр id связывается со значением запроса id. Из-за этого автоматического всязывания методу нет необходимости включать код для получения значения параметра из запроса.



Можно также встроить значение параметра в URL как строку. Например, вместо /Products/Detail?id=3, можно использовать /Products/Detail/3. Стандартное правило маршрутизации имеет /{controller}/{action}/{id}.

Фреймворк MVC поддерживает опциональные аргументы для методов. Опциональные параметры обрабатываются с использованием аргументов типа nullable. Например, если метод принимает данные как часть строки запроса, но вы хотите, чтобы по умолчанию они были равны сегодняшней дате, вы можете использовать код из примера:



Если запрос включает значение для параметра date, то это значение будет передано в метод ShowArticles. Если запрос не содержит значения для этого параметра, то есть аргумент равен null, контроллер может предпринять необходимые действия для обработки этой ситуации.

### 5.2.4 Передача информации в представление

ASP.NET MVC предлагает специальный механизм представлений для генерации внешнего вида страниц (UI). По умолчанию, MVC Framework может использовать типы ViewPage, ViewMasterPage и ViewUserControl наследуемые от существующей страницы aspx WebForms, мастер-страницы (\*.master) и пользовательских элементов управления (\*.ascx).

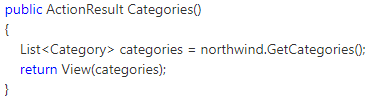
Другой подход – использование специального механизма представлений Razor, созданного специально для ASP.NET MVC, который позволяет значительно уменьшить количество кода необходимого для написания в представлениях. Сегодня, именно Razor рекомендуется и используется в MVC по умолчанию.

В типовом сценарии приложения MVC действия контроллеров обрабатывают входящие запросы. Эти действия используют значения входящих параметров для исполнения кода приложения и для выборки или обновления модели данных из базы данных. Затем методы выбирают представление, которое отображает ответ браузеру.

#### 5.2.4.1 Генерация представлений Модель

В паттерне MVC представления служат для логики пользовательских интерфейсов. Они не должны содержать код бизнес-логики или действий с базой данных. Представление генерирует соответствующий пользовательский интерфейс, основываясь на данных, пришедших из метода контроллера.

Пример генерации представления в контроллере:



В этом примере параметр, передающийся в метод View, является списком объектов Category. Метод View вызывает движок представления, использующий этот список для генерации представления и отображения его в браузере.

#### 5.2.4.2 ViewBag (нет ответа)

#### 5.2.4.3 TempData(нет ответа)

# 6. Разработка представлений Asp.NET MVC(надо отредактировать на ч[то-то подобное](https://docs.microsoft.com/ru-ru/aspnet/mvc/overview/getting-started/introduction/adding-a-view))

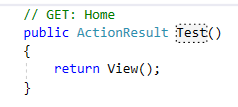
Представление содержит HTML-разметку и контент, отправляемый в браузер. Представление эквивалентно странице.

## 6.1 Создание представлений с помощью движка Razor

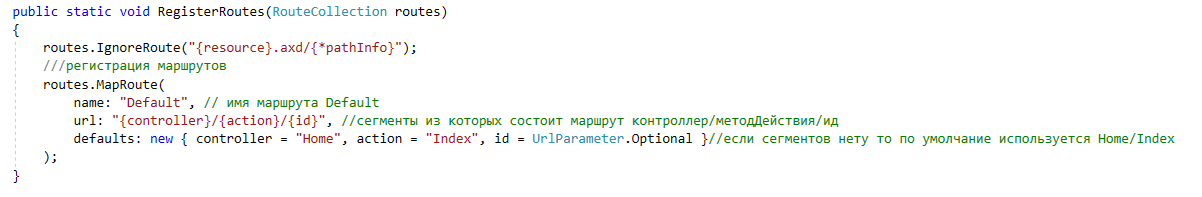
### 6.1.1 Создание нового представления.

Для создания представления нужно кликнуть по методу в классе nameController addView и далее add. В папке Views создастся папка с именем контроллера name, в котором создастся представление с именем метода.

Чтобы получить нужное представление надо в браузере прописать URL <http://localhost:51690/Home/Test> где Home - это имя контролера, а Test метод действия ActionReult



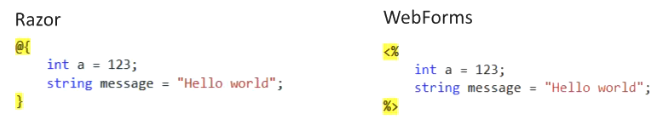
Для лучшего понимания обратимся в RoutConfig.cs именно здесь прописано какой сегмент за что отвечает.



### 6.1.2 Знакомство с Razor синтаксисом.

Razor - новый механизм визуализации в ASP.NET оптимизированный под генерацию HTML разметки написанного на C#, который используется в MVC фрэймворке начиная с третьей версии.

Сравнение Razor и стандартного механизма ASP.NET



Сначала ставим символ @(add) далее скобу и помещаем наш C# код.

HTML Encoded - т.е если внутри переменной message будет находиться HTML разметка она перед тем как отобразиться на HTML странице будет кодирована

пример

HTML Unencoded - данные находящиеся внутри переменной message будут выведены также как они заложены в самой переменной, если там лежит разметка, то она будет визуализироваться браузером в первоначальном виде без кодирования

пример

Комбинирование текста и разметки

Комбинирование кода и текста. Razor не всегда может понять где начинается HTML код поэтому в некоторых ситуациях нужно использоваться @: тогда мы явно указываем где именно находится разметка и текст

Email адреса в разметке. Механизм визуализации распознает email адреса и не воспринимает @ как разделитель кода. В тексте нужно написать @@ для визуализации одного символа @ в текст.

Комментарии устанавливаются как @\* комментарий\*@

• Присоединение представлений к моделям и отображение данных.  
• Понятие модели представления  
• Отличия серверного кода представления от HTML

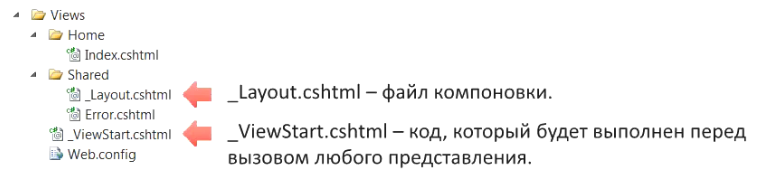
### 6.1.3 Использование HTML Helpers (нет ответа)

• Action Helpers  
• Display Helpers  
• Form Helpers  
• Editor Helpers  
• Validation Helpers  
• Проектирование и реализация своего HTML Helper

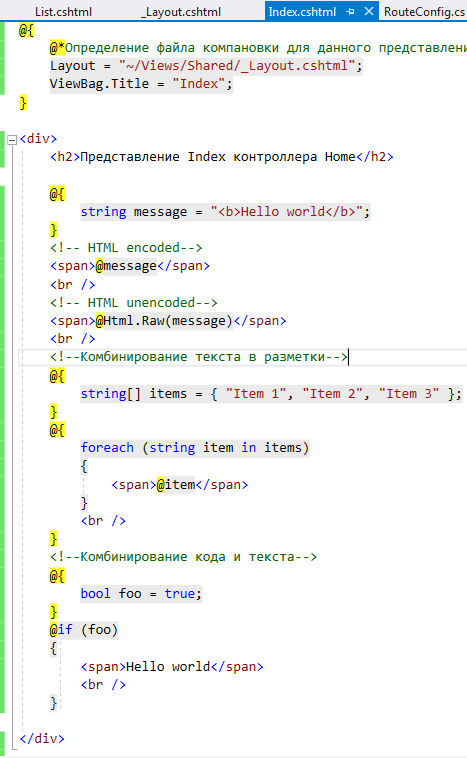
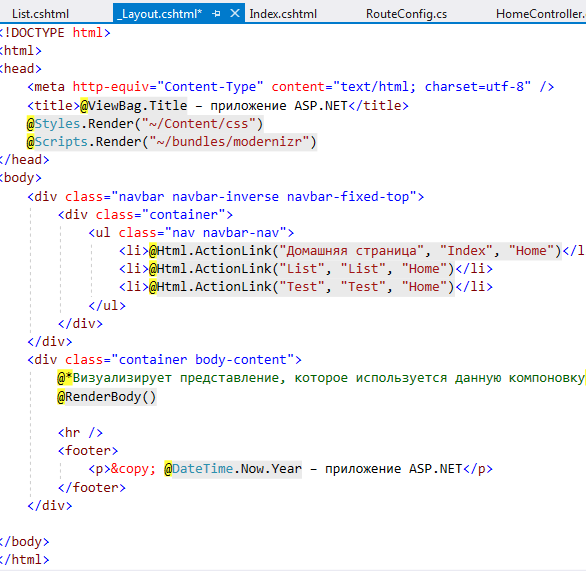
### 6.1.4 Шаблоны элементов представления. Повторное использование кода в представлениях (Мастер-страница)

Компоновка (\_Layout) - эквивалент мастер-страницы в Web Forms. Это шаблон страницы, куда добавляется информация из представления, которое должно быть возвращено после обработки запроса. Мастер-страница - это шаблонная страница в которой находятся полноценная HTML разметка, но так же в этой страницы находятся определенные метки, которые говорят, что сюда необходимо вставить определенный контент, какой именно контент, то за это отвечает конкретное представление.

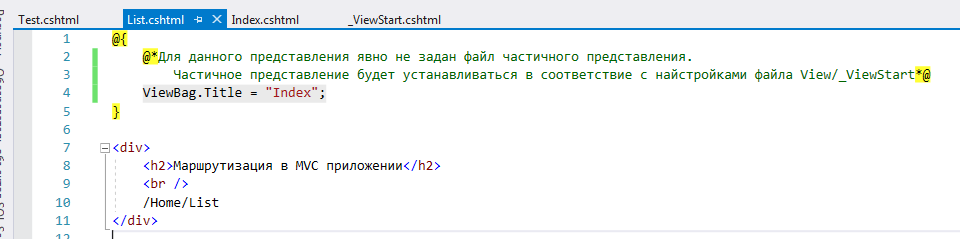
Компоновка используется, чтобы избежать дублирования кода т.е создание единого шаблона сайта, в который можно будет помещать данные из других представлений.



#### 6.1.4.1 Создание и использование шаблонов.

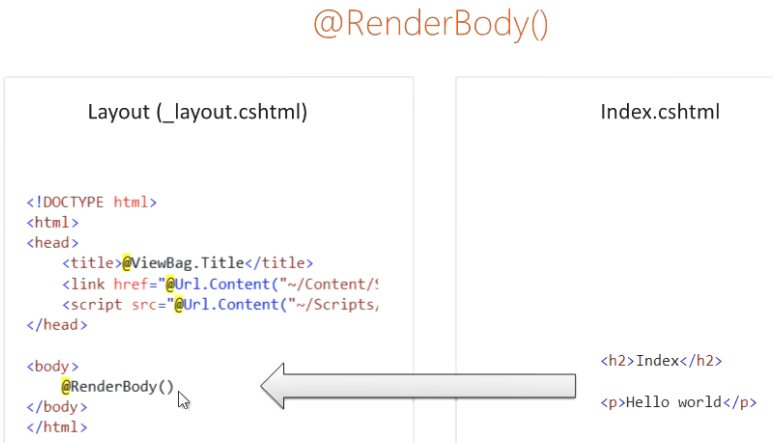
Для использования частичного представления необходимо задать свойство \_Layout для представления и сделать так чтобы в представление была не полная страница, а только та HTML разметка, которую необходимо поместить в частичное представление.

#### 6.1.4.2 Использование ViewStart для отображения частичного представления



### 6.1.5 Использование RenderAction()

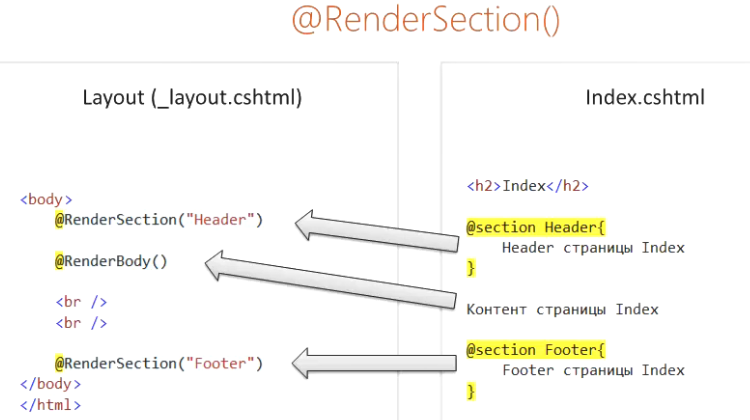
#### 6.1.5.1 RenderBody()



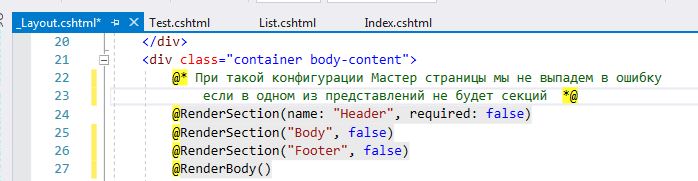
На вид это обычное представление за одним исключением - вызова метода @RenderBody(). Этот вызов является плейсхолдером, на место которого другие представления, которые используют эту мастер-страницу, будут подставлять свое содержимое. И таким образом, мы можем легко установить для представлений веб-приложения единообразный стиль.

Но мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое, поэтому на помощь приходит RenderSection

#### 6.1.5.2 RenderSection()

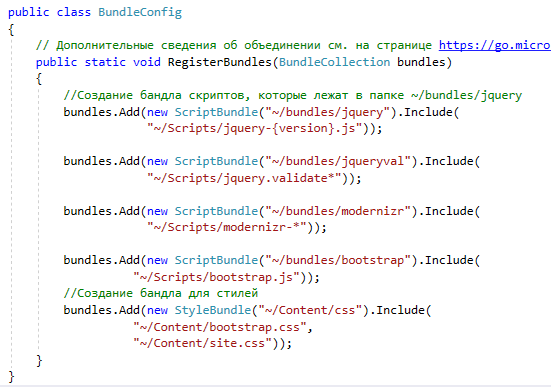


В мастер-странице теперь появляются RenderSection секции Header and Footer, в эти части мастер-страницы будут вставляться контент, находящийся в представлениях в соответствующих блоках HTML кода. В представлениях мы можем использовать ключевое слово @section name{body}

@RenderSection(name: "Header", required: false}

### 6.1.6 Слои и темы(Bundles)

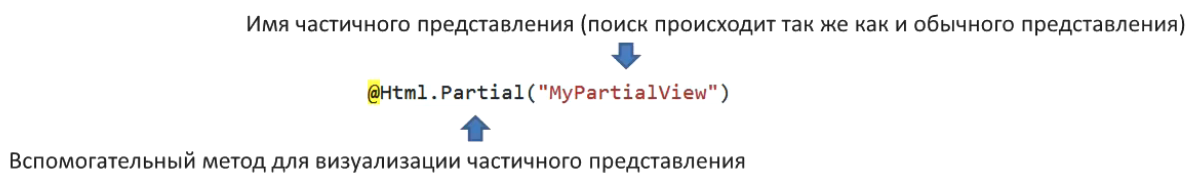
Bundle - это объект, который мы на стороне сервера настраиваем, и в этот объект помещаем ссылки на интерисующие нас JS andCSS файлы, грубо говоря мы создоем мешок, в который мы помещаем нужный для нас сценарий,потом в момент визализации пользовательского интерфейса мы говорим, что хотим взять и в этой часте документа визуализировать определенный бандл. MVC генерирует для нас ссылку, которая ведет на сервер, на сервере находится служба т.е handler которая выдает нам в ответ все те файлы которые находятся в bundle. т.е если раньше у нас было 10 js файлов, мы брали эти 10 файлов и делали 10 отльеных тэков скрипт, что бы эти файлы были скачены по отдельности бразером, но это не выходно т.е надо делать 10 запросов на каждый скрипт. На много выгоднее склееить все файлы в одно целое и даше минифицировать все файлы т.е убрать лишнии комментарии, переименовать длинных методов во что-то более короткое и удобное для браузера, но неудобное для человека. При использования бандлинга все это и происходит  
склеивание и минификация файлов, когда мы загружаем наше приложение то браузеру достаточно сделать один запрос к бандлу, чтобы получить от него весь js код или весь css код в одномответе. Когда мы настраиваем файлы это происходит в файле BundleConfig. В методе RegisterBundles который принимает коллекцию bundles

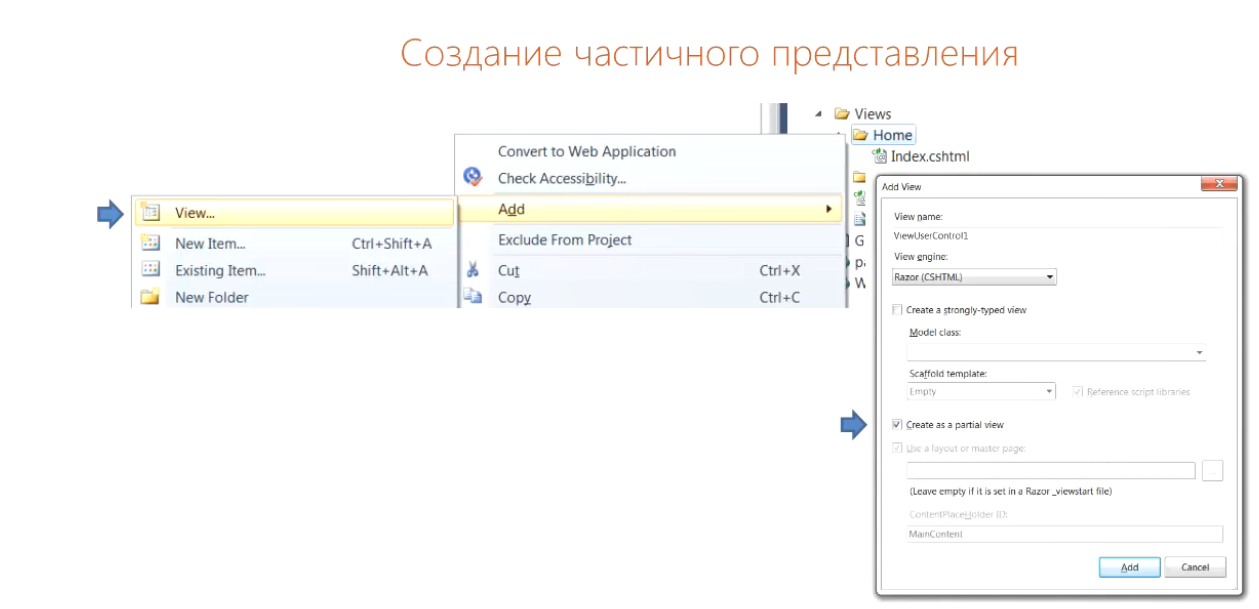


Для включения минификации нужно запускать приложения не в debug, поменяв строчку <compilation debug="false" targetFramework="4.6.1"/> в файле Web.config.

### 6.1.7 Частичное представление(Partial Views)

Частичное представление - это отдельные файлы, содержащие фрагменты разметки, которые могут быть включены в другие представления. Визуализация частичного представления не вызывает метода действия.





# 7. Маршрутизация

## 7.1 Введение в маршрутизацию MVC (Routing)

Функции системы маршрутизации:

Исследование входящих запросов URL. Определение для какого контроллера и действия предназначен запрос. Т.е анализ сегментов, которые передаются пользователем в URL запросе, и в соответствие с этим анализом в нашем приложении запускается соответствующий контроллер и метод действия.

Генерация исходящих URL. Создание URL для ссылок в HTML разметке. Т.е создание обычных HTML ссылок и создание правильных атрибутов href, которые будут ссылаться на корректные методы в контроллерах, которые присутствуют в нашем приложении.  
 Основная задача по маршрутизации ложится на HTTP модуль, который срабатывает в начале обработки каждого запроса приходящему к MVC приложению. Этот модуль работает с настройками, которые мы задаем в Global.asax файле, при запуске нашего приложения

## 7.2 Введение в шаблоны URL

Система маршрутизации работает с использованием набора маршрутов. Эти маршруты вместе образуют схему URL, или схему для приложения, представляющую собой набор URL, которые приложение будет распознавать и реагировать на них.

URL могут быть разбиты на сегменты. Сегменты - это части URL за исключением имени хоста и строки запроса, которые отделяются друг от друга символом /.



### 7.2.1 Переменные сегментов

Система маршрутизации должна сопоставить URL с шаблоном и передать значение сегментов в переменные сегментов.

Переменные сегментов выражаются символами { и }

Шаблон URL с переменными: {controller}/{action}

этот шаблон будет срабатывать в том случае если к приложение будет приходить запрос состоящий из двух сегментов

Переменные сегментов URL запроса

<http://localhost:64399/Admin/Index> controller = Admin, action = Index

[http://localhost:64399/Hello/](http://localhost:64399/Index/Admin)World controller = Hello, action = World

[http://localhost:64399/Admin/](http://localhost:64399/Apples/Oranges)Index/Test Соответствия нет - сегментов слишком много

<http://localhost:64399/Admin> Соответствия нет - сегментов слишком мало

### 7.2.2 Ключевое поведение шаблонов

Шаблоны URL являются консервативными, и будут совпадать только с теми URL, которые имеют то же самое количество сегментов, что и шаблон. Это можно наблюдать в третьем и четвертом примерах в таблице.

Шаблоны URL являются либеральными. Если URL имеет правильное количество сегментов, шаблон извлечет значение для переменной сегмента, каким бы оно ни было.

* + Создание простого маршрута
  + Создание сложного маршрута с использованием специальных переменных сегментов.
  + Маршрутизация с помощью атрибутов.
  + Основы зон.
  + Маршрутизация с использованием зон.

[**Фильтры**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Понятие фильтра.
  + Использование фильтра аутентификации.
  + Использование фильтра авторизации.
  + Использование фильтра действий.
  + Использование фильтра результата.
  + Использование фильтра исключений.
  + Создание своего фильтра.

[**Привязка моделей**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Понятие привязки моделей.
  + Использование стандартной привязки.
  + Настройка системы привязки моделей.
  + Создание собственной привязки.

[**Безопасность в разработке приложений MVC**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Способы взлома Web приложений
  + Конфигурирование приложения
  + Защита форм и данных.

• Использование протокола HTTPS.

* + Основы аутентификации и авторизации.
  + Аутентификация с использованием форм.

[**Интеграция JavaScript и MVC**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Рендеринг и исполнение JavaScript кода

• Добавление JavaScript файлов  
• Использование Content Delivery Network для JavaScript библиотек  
• Введение в jQuery  
• Доступ к HTML элементам с помощью jQuery (селекторы)  
• Введение в jQueryUI

* + Использование AJAX и частичного обновления страницы

• Причины использования AJAX  
• Использование AJAX в MVC

# 25 Bootstrap 3 (нет ответа)

* + Введение в Bootstrap
  + Скрипты и стили
  + Сетка
  + Responsive Design
  + Компоненты
  + Типографика
  + Формы
  + Таблицы
  + Кнопки
  + JavaScript
  + Кастомизация

[**Паттерны проектирования**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Понятие паттерна.
  + Знакомство с основными актуальными паттернами.
  + Анти-паттерны.
  + Принципы SOLID.

[**IoC. Dependency Injection (внедрение зависимостей)**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Паттерн IoC.
  + Основы DI.
  + Знакомство с наиболее используемыми DI контейнерами. NInject, Autofac. Castle Windsor.
  + Внедрение зависимостей в MVC.

[**Производительность и оптимизация веб-приложений**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Общее понимание узких мест приложения. Способы диагностики.
  + Многопоточность в веб
  + TPL
  + PLINQ
  + Асинхронное программирование.
  + Профилировка веб приложений. Нагрузочное тестирование.
  + Методы оптимизации и рефакторинга кода.

# 28 ORM. ENTITY FRAMEWORK 6 (нет ответа)

* + Понятие ORM.
  + Основные подходы к проектированию БД: CodeFirst, DatabaseFirst, ModelFirst.
  + Создание и конфигурирование моделей БД. Fluent vs DataAnnotation.
  + DbContext.
  + Механизм миграций.
  + Ленивая загрузка.
  + DbSet и DbEntry. Механизм отслеживания изменений.
  + Тонкая настройка.
  + Оптимизация запросов к БД и тонкости использования EF 6.
  + Стратегии хранения данных: TPT, TPH.
  + Паттерн репозиторий и UnitOfWork.
  + Entity Framework Core. Ключевые отличия.

# 29 **Asp.Net Identity Framework** (нет ответа)

* + Введение в Asp.Net Identity.
  + Ключевые отличия в новой Identity модели.
  + Identity Context.
  + Пользователи и роли.
  + Использование стандартных средств для конфигурирования пользователей и ролей.
  + Понятие Claim.
  + Аутентификация и авторизация.
  + OAuth.
  + Понятие JWT токена и его использование.
  + Использование сторонних сервисов для аутентификации (Google, Facebook, SSO Claims)

# 30 **WEB** API **2** (нет ответа)

* + Введение в архитектуру REST.
  + Стратегия проектирования RESTfull сервиса.
  + ApiController. Ключевые особенности.
  + Средства для тестирования и документирования сервиса. Swagger.

# 31 **OWIN И KATANA** (нет ответа)

* + Введение
  + KATANA
  + OWIN
  + WebServer Console
  + AppFunc
  + Middleware
  + Создание API
  + Миграция на IIS, OWIN и MVC 5

# 31 Библиотека SignalR. Push-уведомления (нет ответа)

* + Введение
  + Web Sockets
  + SignalR
  + Установка и настройка
  + Создание клиентской части
  + Создание серверной части

[**SINGLE PAGE APPLICATION**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Введение
  + Основные JavaScript Frameworks: Angular, React.
  + SSE
  + Серверная архитектура

[**Основы Asp.NET Core**](https://www.it-academy.by/course/asp-net-developer/nd2-razrabotka-prilozheniy-na-asp-net/)

* + Введение в Asp.NET Core
  + Основные отличия от классического MVC приложения.
  + JavaScript и TypeScript в Core приложениях.
  + Использование Nuget, npm и bower.
  + Конфигурирование приложения. Прощай web.config.
  + Тестирование.
  + Расширение фреймворка. Инструменты dotnet.
  + Развертывание приложения Core.
  + Понятие контейнера. Docker.
  + Основы MS Azure.

# Глава 5 IT-Academy

# 1. Первое занятие - 25.09.2019

## 1.1 Начало работы с BitBucket

[Ссылки на видео от тренера (Александр Комаров). После просмотра оказалось, что это видео по работе с Git, а не конкретно с BitBucket.](https://www.youtube.com/watch?v=en6gms6e54Q&list=PLIU76b8Cjem5B3sufBJ_KFTpKkMEvaTQR)

[Вот видео по работе BitBucket + VS](https://www.youtube.com/watch?v=RcLTiqTc9C8)

## 1.2 Начало работы с VisualStudio.

[Часовой ролик на английском языке, в котором рассмотрены начальные аспекты С#](https://www.youtube.com/watch?v=gfkTfcpWqAY)

## 1.3 Литература:

<https://docs.microsoft.com/en-us/dotnet/welcome>

[C# 7.0. Справочник. Полное описание языка](https://www.ozon.ru/context/detail/id/145563645/)

[CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#](https://oz.by/books/more1028671.html)

# 2. Второе занятие - даты нет